2

TECHNICAL MEMORANDUM
TM 405-90
1 DECEMBER 1990

JOINT DATA SYSTEMS
SUPPORT CENTER

SOFTWARE STANDARDS AND PROCEDURES MANUAL FOR THE JNGG GRAPHICS PROGRAM

APPROVED FOR PUBLIC RELEASE

DISTRIBUTION UNLIMITED

RECORD OF CHANGES

Change Number	Dated	Date Entered	Signature of Person Making Change

DCA FORM 65
MAR 87

JOINT DATA SYSTEMS SUPPORT CENTER

Technical Memorandum TM 405-90

1 December 1990

SOFTWARE STANDARDS AND PROCEDURES MANUAL

FOR THE

JNGG GRAPHICS PROGRAM

SUBMITTED BY:

Jeanne L. Muenzen
Acting Chief
Information Systems Branch

APPROVED BY:

Thomas R. Epperson
Deputy Director
NMCS ADP Directorate

ACKNOWLEDGMENT

This Software Standards and Procedures Manual (SSPM) was prepared under the general direction of the Chief, Information Systems Branch (JNGG); the Chief, General Applications Division (JNG); and the Deputy Director, National Military Command System (NMCS) Automated Data Processing (ADP) Directorate (JN).

CONTENTS

Section		Page

Section		Page

Section		Page

Section		Page

Section | Page

ILLUSTRATIONS

ABSTRACT

This Software Standards and Procedures Manual (SSPM) contains the standards, procedures, guidelines, and restrictions to be used in the development of software for the JNGG Graphics Program.

The major section of the SSPM is section 3 (Software Standards and Procedures). That section is divided into seven major subsections. These subsections include software development tools, techniques, and methodologies (paragraph 3.1); critical lower-level Computer Software Component (CSC) and Computer Software Unit (CSU) selection criteria (paragraph 3.2); software development library (paragraph 3.3); software development files (paragraph 3.4); documentation formats for informal tests (paragraph 3.5); design and coding standards (paragraph 3.6); and formal reviews (paragraph 3.7).

The SSPM is a "living" document and will be modified as necessary to ensure that software developed for the JNGG Graphics Program is consistent, maintainable, and supportive of the JDSSC Release Process.

This manual supersedes the Software Standards and Procedures Manual for the Graphic Information Presentation System (GIPSY) (configuration identifier 8719/88-SSPM-115-*) delivered under Contract Number DCA100-87-C-0064 and dated 4 October 1988.

SECTION 1. SCOPE

The following paragraphs define the scope of the JNGG Graphics Program Software Standards and Procedures Manual by providing the identification and purpose of the program and presenting an introduction to the rest of the document.

1.1 Identification

This Software Standards and Procedures Manual (SSPM) contains the standards and procedures to be used during the development of all Computer Software Configuration Items (CSCIs) and/or Subsystems for all projects comprising the JNGG Graphics Program. The pertinent projects are identified in the following subparagraphs.

1.1.1 Graphic Information Presentation System (GIPSY). GIPSY provides a general-purpose graphics capability that operates on the Worldwide Military Command and Control System (WWMCCS) computer--a Honeywell 6080 mainframe. The system supports the data presentation needs of the JDSSC, which includes all of the WWMCCS sites.

It combines the tools of data retrieval; information processing; and tabular, formatted, graphic, and geographic reports into a single, integrated, on-line, interactive system. It is a file-and data-independent system that is driven by a high level user-oriented language. The graphic capabilities are implemented using a device-independent approach that allows the single integrated system to support multiple, dissimilar devices.

GIPSY effectively serves as an information handling system to connect the user's database to a large set of on-line interactive query and report capabilities. GIPSY can even be run in the batch environment to support requirements involving high-volume output.

GIPSY consists of eight separately loadable modules as listed below:

a. Executive Module (GIPSY)

b. Language Processor (SYNTAX)

c. Data Selection (DATSEL)

d. Matrix Generation (MTXGEN)

e. Geographic Displays (GEOMOD)

f. Graphic Displays (DISPLA)

g. Plotter Interface (GMPS)

h. Formatted Report Processing (GDRMOD).

1.1.2 GIPSYmate. GIPSYmate has its roots in the Common User Contract (CUC) awarded to the International Business Machines (IBM) Corporation on 5 October 1984. Through that contract, IBM developed the WWMCCS Information System (WIS) Workstation (WWS) Early Product which is based on the IBM Personal Computer (PC/XT). The WWS Early Product has a resolution of 720 X 350 pixels and is configured with 640 kilobytes (Kbytes) of internal memory, two removable 5 megabyte (Mbyte) hard disks, and one 5.25 inch floppy disk drive capable of storing 360 Kbytes. It is supplied with commercially-available business graphics software (Energraphics) which produces bar, pie, and line charts, but will not satisfy the graphics requirements of the WWMCCS community. A WIS Joint Program Management Office (JPMO) Letter of Technical Support Requirement (TSR) tasked the JDSSC to design and develop an interim graphics interface between the WWS Early Product and the H6000-based Graphic Information Presentation System (GIPSY). This interface, developed for the eight-color version WWS Early Product, provides the WWS Early Product user with access to all GIPSY capabilities via the Honeywell Visual Information Processor (VIP) 7705 bisynchronous protocol.

The Initial Operational Capability (IOC) of the graphics interface was successfully demonstrated to the WIS Joint Program Manager (JPM) on 19 March 1986 and released to the user community as GIPSYmate 1.0, August 1986. As a result of the demonstration, the WIS JPMO tasked the JDSSC to remove certain limitations present in the IOC as well as to provide the WWS Early Product user community with several capabilities that were available for the now defunct WWMCCS Standard Graphics Terminal (WSGT) through the WSGT Intelligent Terminal System (WITS) Briefing Aids subsystem. These capabilities were integrated into GIPSYmate and made available to the users with GIPSYmate Release 2.0, April 1987.

Suppor^ of an additional VIP emulator, the Enhanced Terminal Capability (ETC), produced by ECDSC, USEUCOM, was integrated into GIPSYmate and made available to the users with GIPSYmate Release 3.0, August 1988. Enhancements of a Disk Operating System (DOS) window and an impending alarm timeout notification were added to GIPSYmate and made available with GIPSYmate Release 3.1, April 1989.

The Joint Staff tasked the WIS JPMO with the development of the Joint Operation Planning and Execution System (JOPES). The development of JOPES requires the support of a graphics software system. The WIS JPMO plans to utilize the Graphic Information Presentation System (GIPSY) and GIPSYmate to provide the graphics support for JOPES. In order for GIPSYmate to function in the JOPES environment and exploit the capabilities of the target workstation for JOPES implementation, the GIPSYmate system had to be modernized using Ada as the high order language (HOL) and implementing the Graphical Kernel System (GKS) on the Zenith Z-248 PC/AT.

1.1.3 Terra Plot (TPLOT). TPLOT is a generalized plotting capability for displaying geodetic information on TPLOT-generated maps, pre-printed maps, or other geodetic frames of reference. It is used within the command and control environment to graphically depict unit identification and location, missile trajectory and bomber sortie routing, reconnaissance flight paths, range plotting, and alternative force posture analysis.

TPLOT is a critical part of the support provided by the JDSSC to the elements of the OSD and the Joint Staff in their strategic planning, targeting, and analysis functions. Other DOD activities using TPLOT for similar graphics applications include the North America Air Defense Command (NORAD), Commander in Chief, Pacific (CINCPAC), Defense Intelligence Agency (DIA), Central Intelligence Agency (CIA), the Defense Communications Engineering Center (DCEC), the Air Force Space Data Services Center, and the Tactical Fighter Weapons Center. While the TPLOT software has been distributed to these DOD sites, no software support is provided by the JDSSC. TPLOT is maintained within JDSSC on the HIS 6080 and IBM 3090 platforms. Plot tapes are produced for the CalComp 5845 electrostatic plotter.

1.1.4 Joint Staff Mapping System (JSMS). JSMS emerged from a Joint Staff requirement to produce accurate graphic representations of events in key parts in the world for aiding decision makers at the National Military Command Center (NMCC).

The Joint Staff has acquired the DeLorme Mapping System (DMS) which provides the capability to retrieve vector and raster map data depicting information from various databases. DMS provides the capability to retrieve user data, overlay this data on DMS maps, and output this data on a printer or plotter. DMS may operate in both Tempest and non-Tempest environments. DMS is hosted on graphic workstations and provides vector and raster map data as specified by the Joint Staff. A requirement exists for DMS to be integrated into other Joint Staff systems. As a result, the JDSSC is charged to provide the following:

a. Operational and technical support for the JSMS

b. System integration

c. Develop/procure software programs and device drivers; identification and recommendation of Commercial Off-The-Shelf (COTS) software packages; recommendations for associated standards

d. Hardware configuration requirements and selections

e. COTS graphics packages that meet the needs of the Joint Staff

f. A database management system (DBMS) using Structured Query Language (SQL)

g. Independent Validation and Verification (IV&V) for software and hardware

h. Long-range technical guidance for the JSMS.

The hardware environment for the JSMS is Wang 280T PC and Zenith Z-248 PC/AT microcomputers operating under DOS. The operating environment may be expanded to include other microcomputer systems and other operating systems such as Unix. The mainframe computer system environments for this project include the Honeywell 6080, IBM 3090/180E, VAX 8650, and VAX 11/785.

1.1.5 Mapping and Graphic Information Capability (MAGIC). The MAGIC effort has evolved from and will build upon the modernization of the WWMCCS standard host-based Graphic Information Presentation System (GIPSY) and the Z-248 PC-based modernized GIPSYmate system.

Designed and developed to meet the needs of a new generation of WWMCCS users, MAGIC will be fielded as a resident system on the WWMCCS Workstation (WWS) and will present a menu-based graphical user interface (GUI) to the user that integrates (as transparently as feasible) the Commercial Off-the-Shelf (COTS) packages also resident on that platform. Processing facilities appropriate to both sophisticated and novice users will be supported as well as the ability to access the full range of database types found on the WWMCCS host (H6000). Functionally, the MAGIC user will have the capability to perform data retrieval and manipulation operations, business graphics displays, geographic and geodetic mapping displays, slide show generation, and graphic editing operations. In terms of system configuration, MAGIC is comprised of eight CSCIs. Each CSCI has been defined according to the functionality it performs and the services provided to both the user and each other. The following is a brief description:

a. **Human Interface** - This CSCI functions as the logical hub of all MAGIC processing activities. When a user initiates a MAGIC session, the program menus are presented to the user, and control of program actions begins in the Human Interface CSCI. The Human Interface CSCI also provides context-sensitive help by displaying help dialogue screens at the touch of a key. The three major components that make up this CSCI are global system control, high level system menus, and a help facility.

b. **Data Management** - This CSCI enables the user to access data from user databases located on the WWMCCS host or the WWMCCS Workstation (WWS). Data in the databases is selected by a user or an application CSCI according to a qualification criterion; the resulting data subset can then be manipulated using Data Management's capabilities and presented as user-formatted reports, statistical graphs, and geographic displays.

c. **Business Graphics** - This CSCI enables the MAGIC user to create and display reports. Each report consists of data selected from a previously identified MAGIC internal format subset of a database. This CSCI gives the user the flexibility to display a report in a form most suited to the user's needs, varying from simple, formatted reports to line graphs and pie charts.

d. **Geographic Mapping** - This CSCI provides a powerful imagery tool for recording, calculating, displaying, analyzing, and general understanding of spatial interrelationships of user data. It is designed to have a stand-alone core that performs cartographic functions by interfacing to the DeLorme Mapping System (DMS). The remaining capabilities of this CSCI are designed as shells around overlays of user-identified data with a variety of graphical display options and interactive editing capabilities.

e. **Graphic Editor** - This CSCI provides the user the interactive capability to create or enhance slides. Slides modifiable by the Graphic Editor can be created by MAGIC system. The Graphic Editor also provides the user with slide output functions. Slide creation or modification can be accomplished by editing functions like draw, text, manipulate object, modify object attributes and manage slide for saving, loading, activating, overlaying, and clearing slides.

f. **Slide Show** - This CSCI provides the user with the capability to organize and display previously saved screen images (slides) which are located in an inventory. The user has the capability to organize these slides and create briefings (an ordered group of slides), as well as adding, removing, rearranging, and renaming slides in the inventory. The user may display slides and briefings at the terminal, printer, or plotter.

g. **Internal Processing** - This CSCI provides a multitude of capabilities that have at least one of two properties: it is required by more than one CSCI, or it is hardware dependent. The capabilities identified by these criteria are as follows:

  (1) Allocation, initialization, and updating of the session record file

  (2) File input/output (I/O) functions such as open, close, read, and write operations

  (3) MAGIC environment initialization and cleanup

  (4) Menu/Window Interface Support that handles interaction between applications CSCIs and the user.

h. **Programmer Utilities** - The specific role of this CSCI is to support the MAGIC software by providing necessary development and maintenance tools. These tools provide the means for generating new releases of the MAGIC software and system files. Programmer Utilities does not interface with the other CSCIs in the executable system.

1.2 Purpose

The Secretary of Defense has directed that the Director, Defense Communications Agency (DCA), utilizing the Joint Data Systems Support Center (JDSSC), provide automated data processing (ADP) and related general support to the National Military Command System (NMCS).

The JDSSC is charged with the development, installation, and operation of an ADP support capability at the command centers of the NMCS. The JDSSC has the responsibility to provide analysis, modeling, design, development, enhancement, and documentation of new and existing capabilities; computer programming support, development and maintenance of data files; on-going and ad hoc operational support; computer support, and other related support to the Joint Staff and the Office of the Secretary of Defense (OSD).

The JNGG Graphics Program is required to provide general-purpose graphics display and associated query and retrieval software support. This entails the analysis, design, development, enhancement, and documentation of new and existing capabilities in the areas of interactive computer graphics, batch-based computer graphics, human engineering, system engineering, interactive query and display systems, batch-based query and report preparation system, integration data/graphic systems, and language translators supporting services implied by this list. The objectives of the efforts in the work area are to:

a. Improve customer service

b. Reduce fielded failures and improve productivity

c. Reduce error rates and improve software quality

d. Reduce time cycles by speeding up the software development process

e. Reduce the cost of poor quality

f. Provide for more efficient use of application resources through the use of standards and promote software reusability

g. Provide a useful, general-purpose development environment with tools for specification, design, implementation, testing, and documentation

h. Explore and apply the state-of-the-art graphics and information systems to command and control functions.

As noted in paragraph 1.1, the JNGG Graphics Program is currently comprised of the following projects: Graphics Information Presentation System (GIPSY), Terra Plot (TPLOT) System, GIPSYmate, Mapping and Graphic Information Capability (MAGIC), and Joint Staff Mapping System (JSMS).

1.3 Introduction

This SSPM contains the standards, procedures, guidelines, and restrictions to be used in the software development associated with all projects comprising the JNGG Graphics Program. The standards and procedures specified in this document will be used to ensure uniformity among the CSCIs or subsystems in all projects as they progress through the JDSSC Release Process described in paragraph 3.1.

THIS PAGE INTENTIONALLY LEFT BLANK

SECTION 2. REFERENCED DOCUMENTS

This section contains a listing of all documents used in the preparation of this manual. A detailed bibliography appears as paragraph 4.1. Technical society and technical association specifications and standards are generally available for reference from libraries. They are also distributed among technical groups and using Federal Agencies.

2.1 Government Documents

This paragraph contains a listing of all Government references (standards, manuals, specifications, and ancillary technical documentation) used in the development of this SSPM.

DI-A-3029	Agenda - Design Reviews, Configuration Audits and Demonstrations
DI-E-3118	Minutes of Formal Reviews, Inspections and Audits
DI-MCCR-80011	Software Standards and Procedures Manual Data Item Description
DOD-STD-2167A	Defense System Software Development
DOD-STD-2168	Defense System Software Quality Program
DOD-STD-7935A	DOD Automated Information Systems (AIS) Documentation Standards
FIPS PUB 151	Portable Operating System Interface for Computer Environments (POSIX)
JDSSC PM 1-90	Documentation Standards and Publication Style Manual
JDSSC PM 2-90	Standards and Procedures for Software Projects
JDSSC PM 4-90	Software Metrics Program
JDSSC TM 402-90	FORTRAN Programming Standards
MIL-HDBK-287	A Tailoring Guide for DOD-STD-2167A, A Defense System Software Development
MIL-STD-480B	Configuration Control - Engineering Changes, Deviations and Waivers
MIL-STD-481B	Configuration Control - Engineering Changes (Short Form), Deviations and Waivers

MIL-STD-482A	Configuration Status Accounting Data Elements and Related Features
MIL-STD-483A	Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs
MIL-STD-1521B	Technical Reviews and Audits for Systems, Equipments, and Computer Software
MIT/LCS/TR-368	The X Window System
SEL-87-002	Ada Style Guide
SEL-87-004	Assessing the Ada Design Process and its Implications: A Case Study

2.2 Non-Government Documents

This paragraph contains a listing of all non-Government references (standards, manuals, specifications, and ancillary technical documentation) used in the development of this SSPM.

ANSI X3.124-1985	Graphical Kernel System (GKS) Functional Description
ANSI X3.159-1989	Programming Language C
ANSI/IEEE Std 990-1987	IEEE Recommended Practice for Ada As a Program Design Language
ANSI/MIL-STD-1815A	Reference Manual for the Ada Programming Language

2.3 Other References

This paragraph contains a listing of all outside, non-Government references (books, periodicals, etc.) used in the development of this SSPM.

Booch, Grady	Software Engineering with Ada, Second Edition
Bruce, Phillip and Pederson, Samuel M.	The Software Development Project: Planning and Management
Charette, Robert N.	Software Engineering Environments: Concepts and Technology
Martin, James	Recommended Diagramming Standards for Analysts & Programmers: A Basis for Automation
Plauger P.J. and Brodie, Jim	Standard C

Plum, Thomas	C Programming Guidelines
Plum, Thomas	Reliable Data Structures in C
Pressman, Roger S.	Software Engineering: A Practitioner's Approach, Second Edition

THIS PAGE INTENTIONALLY LEFT BLANK

SECTION 3. SOFTWARE STANDARDS AND PROCEDURES

The following paragraphs define and describe the specific standards and procedures to be used in JNGG's Graphics Program software development.

3.1 Software Development Tools, Techniques, Methodologies

The following subparagraphs identify and describe the specific tools, techniques, and methodologies that will be used in support of the JNGG Graphics Program.

3.1.1 The JDSSC Software Release Process. The JDSSC is implementing a standard release process across all of its software projects. This initiative is the cornerstone of the Center's Software Engineering Improvement Program. The standard release process, along with other elements of the improvement program, are intended to significantly upgrade JDSSC's capabilities to deliver quality software products. This subparagraph presents the functional makeup of the software release concept and identifies the general criteria by which Project Officers structure their software development programs. It addresses:

a. The definition of a release

b. Release planning

c. The activities of a release

d. User participation.

3.1.1.1 Release Definition. All projects within JNGG shall develop software on the basis of a "release." A release is defined as a planned software delivery to implement specific user-approved and documented capabilities. The release requirement pertains to new software development and to software maintenance of existing systems. Project Officers and their primary user (i.e., customer) coordinate the schedule for software releases.

The timing of software releases is normally based on the capabilities to be delivered and the time/resources required to adequately develop and test each capability. Project Officers shall not schedule software releases based on programmatic conditions alone. Quality products can only be produced when time and resources are allocated to proper development, testing, and quality assessment. Conflicts arising between user-directed schedules and Project Officer estimates for adequate development time shall be resolved by the Program Manager if possible. If the conflict still cannot be resolved, first the Branch Chief and, ultimately, the Division Chief will intervene. The resolution of such conflicts must be documented by the Project Officer and coordinated by the intervening manager and user organization. JDSSC does not want to release software that has not been properly produced.

Projects within JNGG shall address release planning and schedule criteria in their SDPs.

3.1.1.2 Release Plan (RP). Each JNGG software release shall be preceded by a Release Plan (RP). The RP is the what, when, and how of every release. It is the schedule and capability agreement between the Project Officer and the primary user. The RP is a user-oriented document, but must be presented at a level of detail necessary to ensure credibility of the plan. It does not have to be elaborate, but it must define precisely the responsibilities and methods for ensuring that the work will be performed successfully. RPs are living documents and will be kept up to date until the release is fielded. JDSSC PM 1-90 outlines the format and content for an RP.

3.1.1.2.1 Content. The minimum elements in a Release Plan are:

a. A summary of the release that can be understood by the customer

b. A list of capabilities (i.e., the Release Capability List (RCL)) whose status can at any time be reported to the customer and JDSSC management

c. A Work Breakdown Structure (WBS) detailed enough to provide product identification at the lowest level required for quality assurance reviews and configuration management

d. A list of discrete milestones that can be tied to products or product activities on the WBS

e. A risk assessment detailing management and technical issues that might impact the capabilities, cost, or schedule for the release

f. A list of key project personnel and their assignments in relation to the WBS.

3.1.1.2.2 Release Success Criteria. Project Officers shall develop and document success criteria for their releases. Success of a release should be based on meeting schedule, cost, and capability projections. The Release Plan must contain the information needed to evaluate success. For example, RCLs could be used to compare capabilities promised to capabilities delivered.

3.1.1.2.3 Product-Oriented Release. A successful project scheduling and tracking capability is based on the concept of a product-oriented process. A product-oriented release process defines each milestone as a delivered product and not as a point in time. A well-defined set of software products (program, documents, and data) is developed as a result of each activity in the process. The products are reviewed and approved before beginning the next activity.

Progress is monitored through the delivery and review of interim or preliminary products. This may include submission of document outlines, abstracts of documents, position papers, and description of key algorithms or concepts. Multiple drafts of documents may be required for adequate review. A single release can be in multiple activities concurrently since different components can be in different activities. A single component can be in one and only one activity at a given time. Only when all products have been completed is the release complete. The concept of percent completion is not application to a product-oriented process.

The basic concepts of software engineering assume a product-oriented process. Software development functions such as checkpoints, quality assurance, and configuration management have no meaning without identifiable products. Software development projects have identified products called Configuration Control Items (CCIs) with the software development process structured around CCIs. Work breakdown structures, configuration management plans, quality assurance tasks, and project status milestones are all focused against CCIs. In short, without a product-oriented process, there is no process.

The JDSSC Standard Release Process is a product-oriented process. This means that the implementation and management of a JDSSC release are in the development and reviews of its identified products.

3.1.1.3 Release Activities and Functions. The JDSSC release concept is based on the principle that a well-managed and cost-effective software development program must have a structure for determining at any given point in time what is being developed, when it is to be delivered, and how the development is progressing. In order to have this structure, the JDSSC policy on a software release requires that: (1) a specific set of requirements be identified, (2) the specific set of requirements be documented in terms of ADP products and activities, (3) the ADP activities be implemented in accordance with a published and up-to-date schedule, (4) the resulting products be verified against the original requirement, and (5) the software be delivered to the user community for operational verification.

This subparagraph addresses the release process flow, the role of documentation, the role of revie , and the role of configuration management.

3.1.1.3.1 Release Process Flow. All JNGG releases will go through seven general activities:

a. Definition

b. Specification

c. Detailed Design

d. Implementation

e. Acceptance Test

f. Release

g. Review.

At specific points within the release activities, "contracts" are developed between users, the Project Officer, and the technical development staff. This introduction of the term "contracts" is to ensure the staff understanding of the concepts of partnerships and represents a commitment by all parties involved in the process.

The following focus areas are provided to assist Project Officers in developing SDPs and transitioning to the standard process:

a. Projects have a primary user, or user community representative, who participates in release planning.

b. Software requirements for additional capabilities (i.e., new capabilities or updates to old ones) are submitted through the customer, formally documented, and controlled. This also applies to capability improvements suggested by the technical development staff.

c. Software discrepancies (e.g., SPCRs) are documented, controlled, and processed like a software upgrade request. This applies to discrepancies discovered by the user community or the technical development staff.

d. All change requests go through a formal cost and impact analysis in order to estimate the resources needed to implement.

e. Capabilities have a documentation trail from initial tasking through operational fielding. Requirements that are too general should be decomposed into basic capabilities.

f. Milestones are associated with discrete products and must be identified so that there can be no doubt as to whether they were achieved.

g. Projects have a configuration management mechanism and a product review process to ensure a structured approach to software activities.

3.1.1.3.2 Role of Documentation. Through many of the software development activities, documentation is the only measurable product subject to review and technical quality assessment. This means that without documentation, there is no practical way a Project Officer can assess the actual schedule and cost status of a project.

Documentation requirements and the standards for documentation are detailed in the JDSSC Documentation Standards and Publication Style Manual (PM 1-90). This procedures manual identifies documentation products associated with the release process and provides guidelines on how to apply the DOD Standards noted in subparagraph 3.1.2 (i.e., DOD-STD-7935A). The requirements for documentation should be established at the beginning of a release. Project documentation must be considered a formal CCI. JDSSC PM 1-90 outlines the activities of a release during which documentation is produced, updated, and finalized.

Project Officers need to use documentation as a success criterion for each release. Clearly, staff members should be rewarded for the production of quality written work (i.e., complete and technically correct). All staff members, both in-house and contractors, must understand that quality documentation, like quality software development, is expected as a normal part of job performance and job performance evaluations.

Having a procedures manual will not produce project documentation. Document needs get satisfied only when Project Officers and their entire technical staff understand the relationship between complete documentation and software development project success. Project Officers should address individual project documentation needs in their SDPs. Older projects (e.g., GIPSY and TPLOT) which do not have the documentation baseline recommended by JDSSC PM 1-90 should address documentation re-engineering in their SDPs.

3.1.1.3.3 Role of Reviews. All JNGG software development projects shall have a formal review process. This process shall be documented in the project's SDP. Reviews of products and schedules are crucial to management visibility and successful software implementation. Reviews are used in support of project's software quality assurance objectives. The better the quality of the reviews, the better the quality of the products.

There are two broad types of reviews which must be addressed for each project: (1) product quality assessment and (2) process quality assessment reviews. Project Officer guidelines for the two types of reviews are presented in the following subparagraphs.

3.1.1.3.3.1 Product Quality Assessment. As the JDSSC focus on software reliability increases, emphasis is being placed more and more strongly on a separate organizational element, detached from the development group, to be responsible for quality and reliability-related concerns. This is especially true for large projects. This organization entity is normally called the software quality assurance (SQA) group. The organizational independence of SQA ensures that the checks and balances necessary for a successful ADP program are inherent in the management structure. Since the ultimate responsibility for the delivered products rests with the technical development staff, the independent SQA provides a separate reporting channel to project management for monitoring development progress.

The SQA element should provide recommendations on whether to accept or reject milestone products. Additionally, the SQA should conduct in-progress audits and reviews. In the event of conflicts between SQA and the technical development staff, resolution should be accomplished at the appropriate management level (e.g., Program Manager).

The broad purpose of SQA is to assure that computer software design, code, associated documentation, and performance comply with stated requirements.

Through quality assessment reviews, errors and omissions can be uncovered early and corrected before their effect cascades through the entire system. Some top priority focus areas for consideration as SQA responsibilities are:

a. Maintains oversight of requirements, specifications, design, software documentation, and test procedures to ensure requirements traceability

b. Reviews software design prior to coding

c. Participates in technical review, evaluates for technical compliance, and audits development activities

d. Ensures adherence to software standards and procedures

e. Monitors testing and test results to ensure that success criteria are satisfied.

3.1.1.3.3.2 Process Quality Assessment. A good JNGG development process will not just happen. First, it must be carefully planned; second, it must be continually monitored to ensure effective implementation and continuous operation. Monitoring is an audit-related function designed to assess both the process and the compliance with the process. The monitoring process must have objectives and must collect information to monitor (i.e., metrics). Guidelines for monitoring objectives and metrics collection are provided in the following subparagraphs.

3.1.1.3.3.2.1 Objectives. Evaluating and improving the process defined by a project's SDP is the single objective of the monitoring function. Following are some top-priority focus-area questions for Project Officers to consider in their process review program:

a. Are errors being detected early in the development cycle?

b. Are the required technical reviews being conducted?

c. Are cost, time, and capability estimates proving to be accurate? If not, where are the problem areas?

d. Can Project Officers and staff members give accurate status reports quickly?

e. Is there a consistent understanding of the SDP across the entire staff?

f. Are feedback reports available? Is the staff participating in the process improvement program? Are users participating?

g. Are quality objectives defined for the project and are they being met?

3.1.1.3.3.2.2 Metrics. Project metrics must be collected and used as an evaluation instrument for monitoring the software development process. The JDSSC has initiated a metrics collection program through the JDSSC Software Metrics Program (PM 4-90). The advantage of using metrics is that Project Officers and their managers are provided a specific number which can be used to compare trends within a single project over a period of time. Project Officers shall address the collection and use of metrics in their SDPs.

3.1.1.3.4 Role of Configuration Management. All JNGG software development projects shall have a formal Configuration Management (CM) Plan. This plan shall be documented in one of two ways: (1) incorporated as Section 7 of the project's SDP, or (2) a stand-alone Software Configuration Management Plan (SCMP) developed in accordance with DID # DI-MCCR-80009. The role of CM is one of maintaining software stability and controlling change throughout the life cycle of the project. This subparagraph provides guidelines and requirements for a project's CM Plan.

3.1.1.3.4.1 Baseline Concept. Successful achievement of CM implies that a project is able at any time to provide the definite version of the software products (e.g., software code, developmental configuration) or any of the intermediate products that define the software (e.g., requirements and specifications). These controlled product items are called baselines. This concept of baseline is fundamental to an effective CM program.

The JDSSC Standard Release Process uses baselining in its software development approach. Baselining of products will provide projects with the following positive attributes:

a. No changes are made after a product is baselined without agreement of all interested parties.

b. The higher threshold for change tends to stabilize products and projects.

c. The controller of the configuration management process (e.g., the project librarian) has, at any time, a definite version of the product.

3.1.1.3.4.2 CM Activities. JNGG project CM plans must contain the activities to identify, baseline, control, and report changes to project products. In addition, the reader of the project's CM plan should be able to identify and understand how the activity is performed. A general explanation of these activities follow:

a. Configuration Identification - the process of defining, collecting, and identifying all products to be controlled; concerned with control of the item, not its technical adequacy or its quality.

b. Configuration Baselining - the process of officially recognizing a particular configuration with all its associated products (e.g., technical documentation); establishes the exact configuration as a control point which then becomes the reference against which all subsequent changes must be formally accounted.

c. Change Control - the process of evaluating, coordinating, and approving (or disapproving) the implementation of changes to an item under baseline control; assures and expedites the implementation of needed changes and prevents unauthorized and unnecessary ones.

d. Configuration Accounting - the activity keeping track of the current configuration status through a Software Development Library (SDL); provides the information to trace the evolution of the current revision from the initial released configuration.

3.1.2 Applicable Development Standards. Although the overall governing process for JDSSC software is guided by the JDSSC Software Release Process described in subparagraph 3.1.1 above, specific implementation of the types of documentation, the amount of documentation, and types and frequency of reviews is a matter of preference on the part of the Project Officer in consultation with the Program Manager.

Currently, there are two software development standards that may be utilized for implementing those items listed above: DOD-STD-2167A and DOD-STD-7935A. Tailoring guidelines for DOD-STD-2167A are also available through MIL-HDBK-287.

A number of corollary standards are available to provide additional guidance on matters related to configuration management (e.g., MIL-STD-480B, MIL-STD-481B, MIL-STD-482A, MIL-STD-483A), software quality assurance (e.g., DOD-STD-2168), and formal review criteria (e.g., MIL-STD-1521B).

3.2 Critical Low-Level CSC and CSU Selection Criteria

Of the three major approaches available for the coding process (top-down, bottom-up, and RAD), only the top-down approach can be expected to require that a number of low-level Computer Software Components (CSCs) and Computer Software Units (CSUs) be deemed "critical" to the overall development and, thus, be needed prior to the point they would normally be available. In other words, some bottom-up coding would be expected in special cases where the absence of these "critical" low-level CSCs or CSUs would have a detrimental effect on the rest of the project.

In order to be classified as a critical low-level CSC or CSU, specific criteria must be met to ensure that only valid program units are so designated:

a. The program unit must be requested to be so designated by a member of the Configuration Control Board (CCB).

b. The program unit must be designated "critical" by the CCB and appear in the CCB Minutes.

c. At least one of the following technical criteria must be satisfied:

(1) The program unit impacts more than one CSCI.

(2) The program unit interfaces with multiple units within a single CSCI and whose delay would cause serious disruption in the CSCI's development schedule.

(3) The program unit interfaces with external systems.

(4) The program unit provides low-level services such as file input/output (I/O), device drivers, and metafile conversions.

(5) The program unit is to be structured as reusable code (e.g., Ada generic or C macro function).

(6) The language to be used is Assembler.

(7) The program unit uses any Ada constructs or capabilities defined in Chapter 13 of ANSI/MIL-STD-1815A.

(8) The program unit employs bit-field operations in the C Programming Language.

3.3 Software Development Library

The software development library (SDL) is a controlled collection of software, documents, and associated tools and procedures to facilitate the orderly development and subsequent support of the software. Within the SDL are kept the completed Software Development Files (SDFs) for each level of development (i.e., CSCI and CSC SDFs). Additional information on how the SDL will support this effort can be found in the appropriate sections of the Software Development Plans (SDPs) for the projects comprising the JNGG Graphics Program.

The SDL for all projects will be implemented in accordance with the SDP guidance pertinent to each project.

3.4 Software Development Files

The Software Development File (SDF) is a record of specific software development activities associated with a program unit. It is established in skeletal form at the start of the program and becomes an important management tool for monitoring progress during software development and testing activities. The SDF also serves as the vehicle by which the software design may be incrementally reviewed by the customer throughout the software development process.

The SDF is the central place for maintaining all the necessary information about a particular unit and will be heavily relied on when generating the "as-built" system documentation included in the final design documents. The SDF will also be used to accomplish the following:

a. Ensure consistent documentation of the program unit testing process.

b. Track any modifications to the original design of the program unit.

c. Document design and coding decisions that might affect the final system capabilities or performance.

d. Reduce time required for program unit familiarization during system maintenance.

e. Provide a vehicle for monitoring project progress on a program-unit basis.

3.4.1 Creation and Maintenance Responsibility. The contractor shall ensure that the development of each CSC and CSCI (or module and subsystem, as appropriate) will be documented in Software Development Files (SDFs). The responsible Project Manager shall ensure that a separate SDF is established and maintained for each CSC or a logically-related group of CSCs, and each CSCI (or modules and subsystems, as appropriate). The SDF will be subject to Configuration Management (CM) and Software Quality Assurance (SQA) control programs as described in the appropriate project's SDP, SCMP (if applicable) and Software Quality Program Plan (SQPP).

Each SDF will be maintained in a loose-leaf binder with tabbed sections using the format and content guidelines discussed in subparagraph 3.4.2. All SDFs will be maintained for the duration of the project's life cycle.

3.4.2 Format and Contents. The following subparagraphs present the format and describe the contents (which must be included directly or by reference) of the SDFs. To reduce duplication, SDFs should not contain information provided in other documents or SDFs. For the purposes of this discussion, "unit" refers to CSC, CSCI, Module, or Subsystem. An example table of contents appears as figure 3-1.

3.4.2.1 Unit Status. This section is numbered as Section 1 and includes the identification of the unit, the project schedule related to the unit, and an activity log of all important project activities that involve the unit. The unit is first identified within the related computer program and module. Any project schedules and procedure networks that relate to this unit are then included and are kept current with scheduled and actual dates. The activity log is an event-oriented log listing the pertinent activities that describe the status of the unit within the development and testing phases. The activity log should describe the activities which relate to the unit, so that an exact status can always be ascertained.

CONTENTS

Figure 3-1. Sample SDF Table of Contents

3.4.2.2 Requirements Specification. This section is numbered as Section 2 and contains a copy (or provides a reference to) the portions of the Software Requirements Specification pertinent to the unit. Any other documentation related to the unit and not included in the Software Design Document (SDD) should be included in this section.

3.4.2.3 Detailed Design. This section is numbered as Section 3 and contains a copy of the portions of the detailed design documentation related to the unit. This documentation is updated to reflect the final design and implementation of the unit. This section of the SDFs will be relied upon heavily when producing the "as-built" computer program documentation.

The information in this section should include the following:

a. Position. Position of unit within the system hierarchy, including the call sequence and the required parameters

b. Interfaces. Data flow in and out of the unit and interfaces with other software units and the external environment (hardware interfaces)

c. Processing. Description of the processing performed by the unit (control flow)

d. Limitations. Special conditions or limitations.

3.4.2.4 Operating Instructions. This section is numbered as Section 4 and contains information reflecting the actual user interface with the software subsystem. The wording of user prompts and error messages, as well as the description of the circumstances under which the messages are output, should appear here and be updated as needed.

3.4.2.5 Code. This section is numbered as Section 5 and contains a listing of the developed code along with any computer-generated cross-reference listings or maps. An explanation of the procedure required to run the code, including code and file identifiers, should also be included, along with the information and/or files required to submit the unit to configuration control for testing.

3.4.2.6 Unit Test Plan and Procedures. This section is numbered as Section 6 and contains the test plans and procedures for unit testing that were written by the programmers responsible for the unit. These test plans and procedures are informal, but will serve as the basis for accepting the unit for configuration control and higher levels of testing. The test plan and procedures should test the unit exhaustively and specify the acceptance criteria.

3.4.2.7 Test Results. This section is numbered as Section 7 and contains all results of the unit testing accomplished. Included in the test results should be the test configuration used, the date, and the version of the tested unit.

If the unit failed, this should also be recorded, and the tests should be rerun after the unit is corrected.

3.4.2.8 Deficiency Report Changes. This section is numbered Section 8 and contains a log of the Software Release Incident Reports (Irs) and/or Software Problem/Change Reports (SPCRs) generated against the unit as well as an explanation of the resolution of each.

3.4.2.9 Audits and Reviews. This section is numbered Section 9 and contains a copy of all review and audit reports applicable to the SDF. Whenever a review of the SDF is performed, the review comments and the associated checklist for verification that corrections have been completed are placed in this section. Comments pertaining to the unit generated in design reviews should also be included.

3.4.2.10 Notes. This section is numbered Section 10 and contains the following items:

a. All memoranda and design notes related to the unit

b. A version description log containing an explanation of the capabilities of each version of the unit as well as the IRs and/or SPCRs closed by the versions

c. Any additional notes the programmer wishes to include to use at a later date or to help the maintenance programmer to better understand the unit.

3.4.3 Maintenance Procedures. The contractor shall ensure that the SDFs are maintained on a regular basis. It is the Project Manager's responsibility to see that individual SDF notebooks are reviewed on a weekly basis for completeness and updated whenever necessary. At a minimum each SDF will be updated following each In-Process Review (IPR), design team meeting, or informal meeting with the Government whenever discussions pertaining to that particular SDF are presented.

3.5 Documentation Formats for Informal Tests

During the JDSSC Release Process, informal test procedures and test results will be generated and documented for the Government. This paragraph will detail the documentation format to be utilized as a guide for content. The documentation standards contained in subparagraph 3.6.5 will be utilized as well.

For the sake of simplicity, the contractor shall use the same documentation format for all informal testing deliverables (except the STP generated during a preliminary design phase). Inapplicable sections to a specific deliverable will simply be noted as such. The three applicable deliverables are:

a. The informal CSC integration testing document generated in the detailed design phase

b. The detailed informal test procedures generated in the coding and unit testing phase

c. The informal integration test results generated in the CSC integration and testing phase.

As noted previously, the STP produced during the preliminary design phase will not follow the informal documentation format delineated in this section. Rather, the STP will adhere to DID # DI-MCCR-80014A for content guidelines. A sample table of contents appears as figure 3-2.

3.5.1 Scope of Testing. Numbered as Section 1, this section summarizes specific functional, performance, and internal design characteristics that are to be tested. The testing effort is bounded, criteria for completion of each test phase are described, and schedule constraints are documented.

3.5.2 Test Plan. This section appears as Section 2 and describes the overall strategy for integration. Testing is divided into phases and subphases that address specific functional and information domain characteristics of the software. Each of the phases and subphases should delineate a broad functional category within the software and can generally be related to a specific domain of the program structure.

The following criteria and corresponding tests are applied for all test phases:

a. Interface integrity. Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

b. Functional validity. Tests designed to uncover functional errors are conducted.

c. Information content. Tests design to uncover errors associated with local or global data structures are conducted.

d. Performance. Test designed to verify performance bounds established during software design are conducted.

These criteria and the tests associated with them must appear and be discussed in this section.

Also discussed in this section is a schedule for integration, overhead software, and related topics. Start and end dates for each phase are established and "availability windows" for unit-tested modules are defined. A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort. Finally, test environment and resources are described. Unusual hardware configurations, exotic

CONTENTS

SECTION 1.	SCOPE OF TESTING
SECTION 2.	TEST PLAN
2.1	Test Phases
2.2	Schedule
2.3	Overhead Software
2.4	Environment and Resources
SECTION 3.	TEST PROCEDURE
3.1	Description of Test Phase <x>
3.1.1	Order of Integration
3.1.2	Purpose and Modules To Be Tested
3.1.3	Special Tools or Techniques
3.1.4	Overhead Software Description
3.1.5	Test Case Data
3.2	Expected Results for Test Phase <x>
3.3	Description of Test Phase <y>
3.3.1	.
3.3.2	.
3.3.3	.
SECTION 4.	ACTUAL TEST RESULTS
SECTION 5.	REFERENCES
SECTION 6.	APPENDIXES

Figure 3-2. Sample Informal Test Documentation Table of Contents

simulators, and special test tools or techniques are a few of the many topics that may be discussed here.

3.5.3 Test Procedure. In this section (Section 3), detailed testing procedures are described which are needed to accomplish the test plans described in subparagraph 3.5.2. The order of integration and corresponding tests for each integration phase are described. A listing of all test cases and expected results is also included.

3.5.4 Actual Test Results. In Section 4 of this document, a history of actual test results, problems, or peculiarities is recorded. Information contained in this section can be vital during formal CSC testing.

3.6 Design and Coding Standards

The following subparagraphs present the specific standards to be used for design and coding during the JNGG Graphics Program software development. Specifically defined are: Ada style specifications, C style specifications, FORTRAN programming standards, programming languages and graphics standards, diagramming symbology and standards, and documentation standards.

3.6.1 Ada Style Specifications. The following subparagraphs present the structure, coding, and format guidelines to be utilized in Ada development for all projects comprising the JNGG Graphics Program. These style specifications are written specifically to provide:

a. A clear definition of the structure, coding style, and formats to be used in Ada programmatic structures

b. A specification of the commenting and prologue standards to be used in all Ada source code -- including Program Design Language (PDL)

c. A statement of responsibility as to which individual, if any, possesses authority to grant specific waivers and/or deviations to these guidelines.

3.6.1.1 Structure Guidelines. The following subparagraphs present and discuss general guidelines to be used in the structure utilized for Ada subprograms. These guidelines are grouped into the following topics: subprogram cohesion; packages; visibility; tasks; program structure and compilation issues; exception propagation; generic units; and encapsulation.

3.6.1.1.1 Subprogram Cohesion. A subprogram should perform a single, conceptual action. In other words, it must be "functionally cohesive."

3.6.1.1.2 Packages. Packages allow the specification of groups of logically related entities. In their simplest form, packages specify pools of common object and type declarations. More generally, packages can be used to specify groups of related entities including subprograms that can be called from outside the package, while their inner workings remain concealed and protected from outside users.

3.6.1.1.2.1 Utilization. In order for a package to be truly useful, it must perform one or more of the following purposes which are listed below in order of decreasing desirability:

a. Model an abstract entity (or data type) appropriate to the domain of a problem. This is the strongest use of packages for structuring a program. It corresponds to the requirement of functional cohesion for subprograms (see subparagraph 3.6.1.1.1) and contributes to the goal of making the structure of a program reflect the structure of its problem domain.

b. Collect related type and object declarations which are used together (this kind of package should be used only to provide a common set of declarations for two or more library units). Further, it is better to minimize the declaration of variables in these packages. Overuse of packages of variables results in a FORTRAN COMMON block style program decomposition which defeats the abstraction and information hiding properties of packages (see subparagraph 3.6.1.2.6.2).

c. Group together program units for essential configuration control or visibility reasons. This type of package should be used sparingly since it gives no additional information to a human reader on the structure of the program but might, for example, be used to divide a large program at the top level into subsystems to be developed by separate teams. However, it would be best if these subsystem packages fulfilled at least one of the other two purposes in addition to this one.

Packages should NOT be designed based on the procedural structure of the code which calls them. For example, a group of procedures should not be packaged simply because they are all called at system initialization, or because they are always called in a certain sequence. Such a package is closely coupled to the context in which it is used and is not very understandable, reusable, or maintainable as a unit.

A logical hierarchy of packages should be used to reflect or model levels of abstraction.

3.6.1.1.2.2 Nesting. The nesting of packages in Ada software development should adhere to the following guidelines:

a. Nesting of a package specification inside another package specification should be avoided. When a package provides a good abstraction, it hides the details of its implementation. Nesting one package specification inside another either exposes too much of the internal details of the outer package or indicates that the outer package does not provide a good abstraction in the first place. It is usually better to nest the package specification within the body of the outer package. Specific inner package operations can then be called by outer package operations which are at the appropriate level of abstraction for the outer package.

b. Any perceived need for the nesting of packages must be referred to and approved by the Government Project Officer (approval may not be given by the contractor Project Manager). Upon approval, the following guidelines apply:

(1) Nested package bodies should be separate subunits.

(2) Subprogram bodies within a package should be separate subunits.

(3) Packages should not be nested within subprograms except within the main procedure. A possible exception to this recommendation is when a package has objects of variable size which can be allocated when a procedure is called. For example, suppose some data processing uses a BUFFER package which implements a buffer area of a user-specified size:

```
procedure PROCESS_DATA (BUFFER_SIZE : POSITIVE) is

...

  package body BUFFER is

    type BUFFER_TYPE is array (INTEGER range <>) of DATUM;
    BUFFER_AREA : BUFFER_TYPE (1..BUFFER_SIZE);
      ...
  end BUFFER;
...
```

Note, however, that the nested package cannot be reused outside the context of the procedure. An alternative would be to allocate the buffer using an access type. This would require careful handling of allocation and deallocation, but would result in a more self-contained package.

3.6.1.1.3 Visibility. Structural guidelines for visibility are grouped into two areas of interest: the scope of identifiers and the proper usage of the predefined STANDARD package.

3.6.1.1.3.1 Scope. The scope of identifiers should not extend further than necessary. Where a scope is extended by "with" clauses, these clauses should cover as small a region of text as possible. For example, "with" clauses should be placed only on the subunits that really need them, not on their parents. This promotes information hiding and reduces coupling. It can also result in faster recompilation (due to dependency rules).

3.6.1.1.3.2 The Package STANDARD. The package STANDARD should not be named in a "with" clause.

3.6.1.1.4 Tasks. Tasks are entities in Ada whose executions proceed in parallel. Each task can be considered to be executed by a logical processor of its own. Different tasks proceed independently, except at points where they synchronize.

3.6.1.1.4.1 Utilization. Just as for packages (see subparagraph 3.6.1.1.2.1), it is best to have tasks which model problem domain entities. However, in the case of tasks it is also necessary to have some tasks which solely provide interfaces between other tasks and which handle the other issues of concurrency and parallelism mentioned above. However, the program should be structured around the tasks which represent problem-domain entities. A task should fulfill one or more of the following:

a. Model a concurrent abstract entity appropriate to the problem domain.

b. Serve as an access-controlling or synchronizing agent for other tasks, or otherwise act as an interface between asynchronous tasks.

c. Serve as an interface to asynchronous entities external to the program (e.g., asynchronous I/O, devices, interrupts, etc.).

d. Define concurrent algorithms for faster execution o: multi-processor architecture.

e. Perform an activity which must wait a specified time for an event or have a specific priority.

3.6.1.1.4.2 Nesting. The nesting of tasks in Ada software development should adhere to the following guidelines:

a. Usually, tasks should not be nested within tasks or subprograms, except for the main procedure. It should be noted that a subprogram containing a task cannot return until the task has terminated.

b. Nested task bodies should be separate subunits.

3.6.1.1.4.3 Visibility. When only certain entries of a task are intended to be called by program components outside an enclosing package, it is preferable to hide the task specification in the package body and introduce package procedures which, in turn, call the actual entries. This helps to promote information hiding and strengthens the abstraction of the enclosing package (see subparagraph 3.6.1.1.2.2.a). It also hides the use of tasking within the package. Note, however, that special care must be taken if the task entries are to be called using conditional or timed-entry calls. In this case, the outer package must provide special procedures or procedure parameters.

3.6.1.1.5 Program Structure and Compilation Issues. The overall structure of programs and the compilation issues relevant to style guidelines are described in this section.

3.6.1.1.5.1 Program Units. The compilation units of a program belong to a "program library." A compilation unit defines either a library unit or a secondary unit. A secondary unit is either the separately-compiled proper body of a library unit, or a subunit of another compilation unit. The designator of a separately-compiled subprogram (whether a library unit or a subunit) must be an identifier. Within a program library, the simple names of all library units must be distinct identifiers.

a. Library units should be used to allow configuration control of the high-level functional subsystems of a program and for reusable program units.

b. Nested program units should be used to allow direct access to objects declared in an enclosing scope and to increase the structural hiding of the internal implementation details of an enclosing program unit.

c. Bodies of nested program units should be made separate unless they are small enough not to affect the readability of the enclosing unit. The determination as to whether the unit is small enough rests with the Project Manager.

d. Library units in a package structure are preferable to library units which are subprograms. Library units providing services to the main program should always be packages.

3.6.1.1.5.2 WITH Clauses. A context clause is used to specify the library units whose names are needed within a compilation unit. The "with" context clause is discussed below:

a. No unit should have a "with" clause for a unit it does not need to see directly.

b. If only a small part of a given unit needs access to a library unit, then it should appear as a subunit and have its own "with" clause for that library unit (see subparagraph 3.6.1.1.3.1).

3.6.1.1.5.3 Program Unit Dependencies. The rules defining the order in which units can be compiled are direct consequences of the visibility rules and, in particular, of the fact that any library unit that is mentioned by the context clause of a compilation unit is visible in the compilation unit.

A compilation unit must be compiled after all library units named by its context clause. A secondary unit that is a subprogram or package body must be compiled after the corresponding library unit. Any subunit of a parent compilation unit must be compiled after the parent compilation unit.

a. Excessive dependencies between compilation units should be avoided, especially the use of complicated networks of "with" clauses.

b. It is preferable to limit program unit dependencies to a tree structure whenever possible.

3.6.1.1.6 Exception Propagation. Exceptions propagated by a program unit should be considered part of the abstraction or function represented by that unit. Therefore, it should only propagate exceptions which are appropriate to that level of abstraction. If necessary, an exception which cannot be handled by a unit at one level of abstraction should be converted into an exception which can be explicitly recognized by the next-higher level. For example, a STACK package should provide a STACK_FULL exception instead of propagating a CONSTRAINT_ERROR. Similarly, a MATRIX_INVERSE function should raise a program-specified MATRIX_IS_SINGULAR exception rather than propagating a NUMERIC_ERROR.

3.6.1.1.7 Generic Units. A generic unit is a program unit that is either a generic subprogram or a generic package. A generic unit is a "template," which is parameterized or not, and from which corresponding (non-generic) subprograms or packages can be obtained. The resulting program units are said to be "instances" of the original generic unit.

3.6.1.1.7.1 Utilization. Generics should be used in situations where there are no equivalent normal programming constructs and, when used, a generic program unit should fulfill one or more of the following purposes:

a. Provide logically equivalent operations on objects of different type.

b. Parameterize a program unit by a subprogram value.

c. Provide a data abstraction required at many points in a program, even if no parameterization is required.

3.6.1.1.7.2 Generic Library Units. Generic units should be library units.

3.6.1.1.7.3 Generic Instantiation. An instance of a generic unit is obtained as the result of a generic instantiation with appropriate generic actual parameters for the generic formal parameters. An instance of a generic subprogram is a subprogram and an instance of a generic package is a package.

a. The most commonly used generic instantiations should be placed in library units.

b. Generic instantiations should be used cautiously within generic units.

3.6.1.1.8 Encapsulation. The following subparagraphs discuss the use of Ada coding structures which limit the portability of developed applications.

Usage of any features discussed in this paragraph require the approval of the contractor Project Manager (final approval by the Government Project Officer).

3.6.1.1.8.1 Representation Clauses and Implementation-Dependent Features. Representation clauses and implementation-dependent features should, if possible, be hidden inside packages which present implementation-independent interfaces to users.

3.6.1.1.8.2 Input-Output. The following issues relevant to input/output (I/O) processes should be utilized such that hardware dependencies are limited:

a. Use of the predefined package LOW_LEVEL_IO procedures should always be encapsulated in packages or tasks.

b. Use of the LOW_LEVEL_IO procedures should be encapsulated in task objects associated with each item of controlled equipment.

c. File management and textual I/O software should be encapsulated in specialized packages with simple interfaces. This should include file interface code, textual formatting code and user interface code. User interface encapsulation can be especially useful when a system must accommodate increasing levels of user interface sophistication or changing user needs over its lifetime. In these cases, it is crucial that details of the implementation of the user interface be hidden so that changes can be made to it without affecting the rest of the system.

Approval for the use of this feature must be obtained from the Government Project Officer (the contractor Project Manager may not give approval).

3.6.1.2 Coding Guidelines. The following subparagraphs define and discuss guidelines to be used in Ada source code generation. These guidelines are grouped into the following topics: lexical elements; declarations and types; names and expressions; statements; subprograms; packages; visibility; tasks; exceptions; generic units; representation clauses and implementation-dependent features; and input-output.

3.6.1.2.1 Lexical Elements. The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate lexical elements. Each lexical element is either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character literal, a string literal, or a comment. Each lexical element must fit on one line, since the end of a line is a separator.

3.6.1.2.1.1 The Package STANDARD. Language words with predefined meanings in package STANDARD should not be redefined.

3.6.1.2.1.2 <u>Comments</u>. Comments should be used to add information for the reader or to highlight sections of code, and should not merely paraphrase the code.

3.6.1.2.2 <u>Declarations and Types</u>. Ada defines several kinds of entities that are declared, either explicitly or implicitly, by declarations. Certain forms of declaration always occur explicitly as part of a basic declaration. These forms are discriminant specifications, component declarations, entry declarations, parameter specifications, generic parameter specifications, and enumeration literal specifications. A loop parameter specification is a form of declaration that occurs only in certain forms of a loop statement. The remaining forms of declaration are implicit: the name of a block, the name of a loop, and a statement label.

A type declaration declares a type. The elaboration of a full type declaration consists of the elaboration of the discriminant part, if any (except in the case of the full type declaration for an incomplete or private type declaration), and of the elaboration of the type definition.

3.6.1.2.2.1 <u>Constants</u>. A declared object is a constant if the reserved word "constant" appears in the object declaration which must then include an explicit initialization. The value of the constant cannot be modified after initialization.

a. An object should be declared constant if its value is intended not to change.

   Declaring an object to be constant clearly signals both the human reader and the compiler the intention that its value will not change. This not only increases readability, it also increases reliability because the compiler will detect any attempt to tamper with the object. Also, it can result in some decrease in executable size and better run-time efficiency.

b. Defining a constant object is preferable to using a numeric literal or expression with constant value, as long as the constant object has an intrinsic conceptual meaning.

   There is no use to defining a constant object when a numeric literal is obviously more appropriate. For example, using "ONE" instead of "1". However, the use of constant objects with intrinsic meaning (such as "BUFFER_SIZE" or "FIELD_OF_VIEW") can greatly increase the readability of code. Further, the code is more maintainable since a change in a value will be localized to the constant declaration.

c. A named number (i.e., PI) should be used only for values that are truly "universal" and "typeless." Other numeric constants should be declared with an explicit type.

Such constants as "PI" and cardinal integers (e.g., a "number of things") should be named numbers. Note also that declaring a constant in terms of a predefined numeric type (INTEGER, FLOAT, etc.) has no advantage over a named number since these predefined types provide only range and accuracy constraints and no additional conceptual meaning. In fact, since the range and accuracy of predefined numeric types is implementation-defined, portability can be increased by using named numbers, in those cases where a constant of a user-defined type is not more appropriate.

```
NUMBER_OF_SENSORS  : constant := 4;  -- This is a named number
MAIN_SENSOR_NUMBER : constant SENSOR_INDEX := 2;
```

3.6.1.2.2.2 Types. Separate types should be used for values that belong to logically independent sets as well as for distinct concepts.

```
type X_COORDINATE is range 1..640;
type Y_COORDINATE is range 1..480;
type PIXEL_VALUE is range 0..255;
type IMAGE_GRID is array (X_COORDINATE, Y_COORDINATE) of PIXEL_VALUE;
```

A data type characterizes a set of values and a set of operations applicable to objects of the type. In the above example, each coordinate has a type because coordinates are independent entities. Explicitly declaring these types makes the concepts more obvious to a human reader and also allows the compiler to detect mistakes such as:

```
image (Y, X) := PIXEL; -- Should be "(X, Y)"
```

The drawback of this kind of typing is that the following construct is illegal:

```
if X = Y then           -- ILLEGAL since X and Y have different types
   ...
```

A type conversion must be used:

```
if X = X_COORDINATE(Y) then
   ...
```

Note that, depending on context (and compiler quality), there may or may not be some run-time penalty associated with type conversion (e.g., testing of range constraints).

3.6.1.2.2.3 Enumeration Types. An enumeration type should always be used in preference to an integer type, unless the logical nature of the concept to be modeled demands the other. For example:

```
type DEVICE_MODE is (READ_ONLY, WRITE_ONLY, READ_WRITE);
```

3.6.1.2.2.4 Floating Types. To enhance portability, the range and accuracy of a floating point type should be specified. The precision for the predefined floating types (FLOAT, etc.) is implementation-dependent, though all implementations should provide at least six decimal digits of accuracy. Explicitly declaring floating point ranges can yield more reliable and more efficient as well as more portable code.

3.6.1.2.2.5 Record Types. A record type should be used instead of an array type even when all the record components have the same type, as long as each component can be sensibly named and the components do not need to be dynamically indexed. The following example is preferable to defining COMPLEX as an array of two FLOATS:

```
type COMPLEX is
  record
    REAL      : FLOAT;
    IMAGINARY : FLOAT;
  end record;
```

Overcomplicated record structures should be avoided by grouping related data into subrecord types. For example:

```
type COORDINATE is
  record
    ROW    : FLOAT;
    COLUMN : FLOAT;
  end record;

type WINDOW is
  record
    TOP_LEFT     : COORDINATE;
    BOTTOM_RIGHT : COORDINATE;
  end record;
```

Enumeration types should be used for discriminants of record variants whenever possible. A discriminant should have a default initialization only if the discriminant value is intended to change over the lifetime of an object.

3.6.1.2.2.6 Access Types. Access types should not be used when static types and stack allocation would be sufficient. They should be used only when it is necessary to have data structures with dynamic pointers or to dynamically create objects. However, access types may be needed for static objects if this leads to a more consistent programming style (e.g., so that similar static and dynamic objects are treated identically). For example, if linked lists are used in a program, there may be some lists which are constant, but which would allow passing these constant lists to subprograms which also handle dynamic lists.

3.6.1.2.2.7 Object Declarations. Each object declaration should declare only one object. For example, the following objects should be declared in separate declarations even though they are all of the same type:

```
TABLE_SIZE    : TABLE_RANGE;
TABLE_INDEX   : TABLE_RANGE;
CURRENT_ENTRY : TABLE_RANGE;
```

An object should not be declared using an unnamed constrained array definition. The unnamed array definition is the only case in Ada where an object can be declared to be of a type which does not have a name. Instead, the array type should be named in an array definition, and that name used in the object declaration, even if there is only one object declared of that type.

```
type POOL_TYPE is array (POOL_RANGE) of CHARACTER;

POOL : POOL_TYPE;
```

Objects should be initialized. Where possible, objects should always be initialized by their declaration, rather than in later code.

```
IS_FOUND : BOOLEAN := FALSE;
```

3.6.1.2.3 Names and Expressions. Names denote declared entities, whether declared explicitly or implicitly. Names can also denote objects designated by access values; subcomponents and slices of objects and values; single entries, entry families, and entries in *families of entries*. Finally, names can denote attributes of any of the foregoing.

An expression is a formula that defines the computation of a value. The type of an expression depends only on the type of its constituents and on the operators applied. For an overloaded constituent or operator, the determination of the constituent type, or the identification of the appropriate operator, depends on the context.

3.6.1.2.3.1 Aggregates. An aggregate is a basic operation (group of component associations) that combines component values into a composite value of a record or array type. Each component association associates an expression with components (possibly none). A component association is "named" if the components are specified explicitly by choices. Otherwise, it is called "positional." For a positional association, each component is implicitly specified by position--in the order of the corresponding component declarations for record components, in index order for array components.

a. Aggregates are preferable to individually setting all or most of the components of an array or record.

b. Named aggregates, rather than positional aggregates, should be used wherever possible.

c. The "others" choice should not be used within aggregates without good reason. Individual cases must be referred to the contractor Project Manager for approval (final approval given by the Government Project Officer).

3.6.1.2.3.2 Static Expressions. Where possible, universal expressions are preferable to static (but not universal) expressions, which are in turn preferable to dynamic expressions.

Since they do not depend on run-time dynamics, static expressions are easier for a human reader to understand. Also, universal expressions maximize accuracy and portability, and static expressions eliminate run-time overhead.

3.6.1.2.3.3 Short-Circuit Control. Short-circuit control forms should be used only to avoid evaluation of an undefined or illegal expression and not to merely optimize execution. Use of short-circuit forms must be approved by the Government Project Officer (approval may not be given by the contractor Project Manager).

```
(N /= 0) and then (TOTAL/N > LIMIT)
(INDEX = 0) or else USER(INDEX).NOT_AVAILABLE
```

The short-circuit control forms should be used to signal to a human reader that the correctness of the second condition depends on the results of the first. They should not be used for micro-efficiency reasons, concerns better handled by an optimizing compiler. If efficiency considerations are substantially important, "if" statements should be used instead of the short-circuit forms with functions used to avoid repeated code, if necessary.

3.6.1.2.3.4 Type Qualification. An explicit type conversion should not be used if a type-qualified expression is meant:

```
Good:   LONG_FLOAT'(3.14159)
Bad:    LONG_FLOAT (3.14159)
```

A qualified expression is used to state explicitly the type, and possibly subtype, of a value. A type conversion, however, results in the dynamic conversion of a value to a target type. Sometimes a type conversion can be used to serve the purpose of a type qualification. However, if the operand is already of the desired base type, a conversion is not really necessary and a qualification should be used instead.

Situations where type qualification is necessary should be avoided if possible. Other than where absolutely necessary, type qualification may be justified only if it makes the program clearer to a reader. The main case to avoid is when the type of an enumeration literal or aggregate is not known from context. For example:

```
type COLOR is (BLACK, RED, GREEN, BLUE, WHITE);
type LIGHT is (RED, YELLOW, GREEN);
```

```
procedure SET (COLOR_CODE : in COLOR);
procedure SET (COLOR_CODE : in LIGHT);
...
...
...
SET (COLOR'(RED));    -- Type qualification must be used here to
SET (LIGHT'(RED));    -- resolve the overloading of SET and RED.
```

In this case, it would be better to rename one of the SET procedures, or to at least give them different parameter names so the overloading could be resolved using named notation.

3.6.1.2.4 Statements. A statement defines an action to be performed and is either simple or compound. A simple statement encloses no other statement. A compound statement can enclose simple statements and other compound statements.

3.6.1.2.4.1 Slice Statements. Array slice assignments should be used rather than loops, to copy all or part of an array. This is more readable and less error prone, especially in the case of slices with overlapping ranges.

```
CLIENT_LIST (LAST_CLIENT..NUMBER_OF_CLIENTS)
    := NEW_CLIENTS (1..NUM_NEW_CLIENTS);
```

3.6.1.2.4.2 IF Statements. An "if" statement should not be used to create the effect of a "case" statement controlled by the value of an enumeration type other than BOOLEAN.

3.6.1.2.4.3 CASE Statements. A "case" statement should not be controlled by a BOOLEAN value and, when possible, the explicit listing of all choices on a "case" statement is preferable to the use of an "others" clause. This makes it easier for a human reader to see that the proper actions are being taken in all cases. Further, if the enumeration type of the control expression is modified, the compiler will indicate overlooked alternatives. However, there are cases when an "others" clause makes sense. For example, if the control expression is of type CHARACTER, then it is usually best to use an "others" clause to handle the "undesired characters" case.

3.6.1.2.4.4 Block Statements. Blocks should be used cautiously to introduce local declarations or to define a local exception handler. To some extent, a block can be thought of as a procedure which is hard-coded in-line. However, a procedure call contributes to readability precisely by not having its source code in-line (providing a "functional abstraction"). Therefore, blocks should always be used cautiously and only for specific purposes. Thought should always be given to using a procedure call instead of a block to improve readability. Declarations of objects used only within a block should be nested within the block. Block usage must be approved by the contractor Project Manager (final approval by the Government Project Officer).

3.6.1.2.4.5 <u>EXIT Statements</u>. "Exit" statements should be used cautiously and only when they significantly enhance the readability of the code. It is often more readable to use "exit" than to try to add BOOLEAN variables to a "while" loop condition to simulate exits from the middle of a loop. However, it can be difficult to understand a program where loops can be exited from multiple places. If possible, it is best to limit the use of "exit" statements to one per loop and it is more readable still to use "exit when". Use "if...then...exit, end if;" when "last wishes" processing is needed.

3.6.1.2.4.6 <u>RETURN Statements</u>. It is preferable to minimize the number of return points from a subprogram, as long as this does not distract from the natural structure or readability of the subprogram.

3.6.1.2.4.7 <u>GOTO Statements</u>. Neither "goto" statements nor labels should ever be used. Use of the "goto" makes the textual structure of code less reflective of its logical structure. Possible uses of the "goto" statement can always be handled by other constructs in Ada. Cases in Ada when the "goto" still seems appropriate almost always indicate poorly designed code. It is better to redesign the code than to use the "goto" statement.

3.6.1.2.5 <u>Subprograms</u>. A subprogram is a program unit whose execution is invoked by a subprogram call. There are two forms of subprograms: procedures and functions.

3.6.1.2.5.1 <u>Parameters</u>. The following guidelines should be utilized with subprogram parameter specifications:

a. Subprograms with equivalent parameters should generally declare each parameter in the same position with the same identifier.

b. Parameters with default expressions should usually be used only when they have very well known default values and/or they are defaulted most of the time and the default is only overridden in special circumstances.

c. Parameters with default expressions should generally be placed at the end of the parameter list, so that they may be omitted if desired in calls using positional notation.

3.6.1.2.5.2 <u>Recursion</u>. A recursive subprogram should be used <u>only</u> if two conditions exist: (1) it is conceptually simpler for a given problem than a corresponding iterative subprogram and (2) a waiver has been obtained from the contractor Project Manager (final approval for the waiver is obtained from the Government Project Officer).

3.6.1.2.5.3 <u>Functions</u>. A subprogram without side-effects returning a single value should generally be written as a function. Since functions can be called from within expressions, there is more freedom in how a function can be used. For example, if a function is to be called only once within some other subprogram, it can be used to initialize a constant object.

```
procedure PROCESS_SENSOR_DATA is

  MAIN_SENSOR_DATA : constant SENSOR_DATA
      := READ_SENSOR (MAIN_SENSOR_INDEX);

begin -- PROCESS_SENSOR_DATA
  ...
```

However, if this sort of freedom is specifically not desired, or if a subprogram has side effects, then use of a procedure should be considered instead of a function, even if the subprogram returns only a single value.

3.6.1.2.5.4 Overloading. Overloading of subprograms should not be used except in the following cases:

a. Widely used utility subprograms which perform identical or very similar actions on arguments of different types (e.g., square-root of integer and real arguments)

b. Overloading of operator symbols.

Note that this is not meant to cover subprograms with identical names in different packages, unless both subprograms are visible through "use" clauses for their packages.

Operator symbols should be overloaded only when the new operator definitions comply closely with the traditional meaning of the operator (e.g., "+" for vector addition). For example, "+" might be used for vector addition, but should certainly not be used for vector dot product.

3.6.1.2.6 Packages. Packages allow the specification of groups of logically related entities. In their simplest form, packages specify pools of common object and type declarations. More generally, packages can be used to specify groups of related entities including subprograms that can be called from outside the package, while their inner workings remain concealed and protected from outside users.

3.6.1.2.6.1 Initialization. Calls from the initialization statements of a package to subprograms outside the package should be avoided.

3.6.1.2.6.2 Visible Variables. Variable declarations in package specifications should be minimized. The use of variables in a package specification reduces the abstraction and information hiding properties of that package. For example, a variable cannot provide protection against being changed by units other than the package. Therefore, it is better to use a function rather than a variable to read data from a package. It is also better to use a procedure rather than a variable to give data to a package, since a variable cannot trigger any package operations and a variable declaration often exposes some internal data representation details of the package.

The private part of a package specification should only be used to supply the full definitions of private types and deferred constants; all other declarations should be put in the package body.

If possible, objects of private type should be initialized by default.

3.6.1.2.7 Visibility. The concept of visibility within Ada is very important. The visibility rules determine which library units can "see" as well as those that can "be seen." For this purpose, the "use" context clause as well as dot notation are the two main tools available.

3.6.1.2.7.1 The USE Clause. Contractor Project Manager approval is required for utilization of the "use" clause in all cases except for the predefined package TEXT_IO (final approval by the Government Project Officer). Upon approval, the following guidelines for usage apply:

a. For packages of commonly known utility operations used throughout a program (e.g., MATHLIB)

b. To make overloaded operators visible, so that they may be used in infix notation

c. For predefined input/output packages (e.g., TEXT_IO, instantiations of INTEGER_IO, etc.)

d. To make enumeration constants visible so that they can be named without using the dot notation.

Note that even when a "use" clause is used, the dot notation should still be used in cases other than those listed above.

3.6.1.2.7.2 Renaming Declarations. For a name with a large number of package qualifications, a renaming declaration may be used to define a new shorter name. The new identifier should still reflect the complete meaning of the full name. In any case, the usage of Ada's renaming capability requires the approval of the contractor Project Manager (final approval by the Government Project Officer).

For a function which can be appropriately represented by an operator symbol name (see subparagraph 3.6.1.2.5.4), a renaming declaration may be used to give it such a name. For example, a MATRIX_MULTIPLY function could be renamed "*".

3.6.1.2.7.3 Redefinition. Items from the package STANDARD should not be redefined or renamed and redefinition of an identifier in different declarations should be avoided.

3.6.1.2.8 Tasks. Tasks are entities in Ada whose executions proceed in parallel. Each task can be considered to be executed by a logical processor of its own. Different tasks proceed independently, except at points where they synchronize.

3.6.1.2.8.1 Task Types. The following rules should be observed when utilizing task types:

a. A task type should be used only when multiple instances of that type are required. Otherwise a directly named task should be used.

b. Identical tasks should be derived from a common task type.

c. Static task structures should be used whenever they are sufficient. Access types to task type should be used only when it is essential to create and destroy tasks dynamically, or to be able to change the names with which they are associated.

3.6.1.2.8.2 Task Termination. A task nested within the main program must terminate by reaching its "end", or must have a selective wait with a terminate alternative. All tasks nested within a program must terminate before the program can terminate. Therefore, if this guideline is not followed, it will be impossible for the main program to ever terminate other than by aborting all nested tasks. However, "abort" statements are to be avoided (see subparagraph 3.6.1.2.8.7).

Tasks dependent on library units should not use the "terminate" alternative of a select statement. Therefore, other provision should be made for the graceful termination of such tasks at system close down. Tasks which are dependent on library units will not terminate due to a "terminate" alternative. Therefore, a library unit task should have an entry which forces termination. If it does not, an "abort" statement in the main program may be used to terminate the task. However, "abort" statements are to be avoided (see subparagraph 3.6.1.2.8.7).

3.6.1.2.8.3 Entries and ACCEPT Statements. The following guidelines must be used when making "entry" calls and using "accept" statements:

a. Only those actions should be included in the "accept" statement which must be completed before the calling task is released from its waiting state.

b. A task should never call its own entries, even by indirection. This would result in a deadlock.

c. Conditional entry calls should be used sparingly to avoid unnecessary busy waiting.

3.6.1.2.8.4 DELAY Statement. A "delay" statement should be used whenever a task must wait for some known duration. A "busy wait" loop should never be used for this purpose.

It is important to remember that "delay t" provides a delay of at least t seconds, but possibly more. A program should not rely on any upper bound for this delay, especially when tasks are used (since tasks must compete for CPU time). The following example shows how to alleviate this problem in a periodic activity:

```
...
NEXT_TIME := CALENDAR.CLOCK + REQUIRED_PERIOD;

PERIODIC_ACTIVITY:
while STILL_TIME loop

  -- Perform activity
  ...
  -- Correct for delay statement incertitude

  PERIOD := NEXT_TIME - CALENDAR.CLOCK;

  if PERIOD < 0.0 then              -- Processing was too slow
    NEXT_TIME := CALENDAR.CLOCK;    -- Avoid cumulative effect
  end if;

  NEXT_TIME := NEXT_TIME + REQUIRED_PERIOD;

  delay PERIOD;

end loop PERIODIC_ACTIVITY;
```

The "delay" statement should normally only be used to manage interaction with some external process which works in real time, or to create a task which behaves in a well-defined manner in real time.

3.6.1.2.8.5 Task Synchronization. Knowledge of the execution pattern of tasks (e.g., fixed, known time pattern, etc.) should not be used to avoid the use of explicit task synchronization.

3.6.1.2.8.6 Priorities. Only a small number of priority levels should be used. The priority levels used should be spread over the range made available to type PRIORITY in the implementation. Names should be given to the priority levels by declaring constants of predefined type PRIORITY and grouping these declarations into a single package.

Using only a small number of priority levels makes the interaction of the various prioritized tasks easier to understand. On the other hand, spreading the levels across the available range allows easy insertion of a new level between existing levels if this later becomes necessary. As with other literal numbers, the use of names is more readable than the use of the literals. Further, for priorities, the allowable range of levels is implementation-dependent. Naming priority levels by constant declarations grouped into a single package restricts the implementation-dependency to that package. For example:

```
with SYSTEM;
package PRIORITY_LEVELS is

  LOWEST     : constant SYSTEM.PRIORITY := SYSTEM.PRIORITY'first;
  HIGHEST    : constant SYSTEM.PRIORITY := SYSTEM.PRIORITY'last;
  NUMBER     : constant HIGHEST - LOWEST + 1;
  AVERAGE    : constant SYSTEM.PRIORITY := NUMBER / 2;
  IDLE       : constant SYSTEM.PRIORITY := LOWEST;
  BACKGROUND : constant SYSTEM.PRIORITY := AVERAGE - 20;
  USER       : constant SYSTEM.PRIORITY := AVERAGE - 10;
  FOREGROUND : constant SYSTEM.PRIORITY := AVERAGE + 10;

end PRIORITY_LEVELS;
```

For any group of related tasks, such as those declared within the same program unit, priorities should be specified either for all, or for none of them. This avoids confusion about the scheduling of tasks with undefined priorities.

3.6.1.2.8.7 <u>ABORT Statements</u>. Abortion of tasks should generally be avoided. Aborting a task can produce unpredictable results. In particular, do not assume anything about the moment at which an aborted task becomes terminated. The "abort" statement should be used only in case of unrecoverable failure.

3.6.1.2.8.8 <u>Shared Variables</u>. When "sharing" variables between tasks, the following guidelines must be utilized:

a. Tasks should not directly share variables unless only one of them can possibly be running at any one time.

b. Any task which uses shared variables should identify in its documentary comments all the shared variables that it uses.

3.6.1.2.8.9 <u>Local Exception Handling</u>. To allow the handling of local exceptions without task termination, a task should generally have a block statement with an exception handler coded within its main loop.

```
begin -- SOME_TASK

  MAIN_LOOP:
  loop

    LOCAL:
    begin -- LOCAL

      -- Task code for this block
      ...
    exception
```

```
          ... handle local exceptions ...

        end LOCAL;
      end MAIN_LOOP;

    exception
      ... handle fatal exceptions ...

    end SOME_TASK;
```

3.6.1.2.9 Exceptions. The Ada facilities for handling errors or other exceptional situations that arise during program execution are called exceptions or exception handlers.

3.6.1.2.9.1 Utilization. An exception should be used only for one or more of the following reasons:

a. It reports an irregular event which is outside the normal operation of a program unit or in some sense an error.

b. It is used where it can be argued that it is safer (more defensive) than the alternative, in particular to guard against omissions of error checking code for especially harmful errors.

c. It reports an event for which it is inconvenient or unnatural to test at the point of cause/occurrence and thus use of the exception enhances readability. Exceptions declared in package specifications are really part of the abstraction defined by that package. Therefore, their use should be integral to the design of the package (see subparagraph 3.6.1.1.5.1).

Also, note that the predefined exceptions should be used with care. Due to allowable implementation differences, they should not be relied upon to indicate particular circumstances.

Exceptions should not be used as a means of returning normal state information. For example, a STACK package may have STACK_FULL and STACK_EMPTY exceptions which are raised by its PUSH and POP subprograms. However, these subprograms should not be used solely to raise exceptions to test if the appropriate conditions are true. Instead, the package should provide BOOLEAN functions such as FULL and EMPTY to test for these state conditions.

3.6.1.2.9.2 Exception Handlers. The following rules apply when using exception handlers in Ada programs:

a. The exception handler choice "others" should be used only if it is necessary to ensure that no UNANTICIPATED exception can be propagated or if some special action must be taken before propagation. For example, important tasks should generally have an "others" clause in a local exception handler (see subparagraph 3.6.1.2.8.9) to prevent them from terminating due to unanticipated exceptions. However, in the case when it can be expected that a certain exception may sometimes occur, then that exception should always be explicitly named in the exception handler.

b. Recursion should not be used within an exception handler.

c. Exception handlers on block statements should be used sparingly. One of the advantages of using exceptions is that it separates the error handling code from the more often executed normal-processing code. Excessive use of exception handlers in block statements can defeat this advantage.

3.6.1.2.9.3 RAISE Statements. A "raise" statement raises an exception. For the execution of a raise statement with an exception name, the named exception is raised. A raise statement without an exception name is only allowed within an exception handler. It raises again the exception that caused transfer to the innermost enclosing handler.

a. Exceptions declared in the specification of a package which represents a problem domain entity should not be raised outside that package. Exceptions declared in a package specification should be considered part of the abstraction defined by that package. These exceptions provide special "signals" from the package operations, and thus should not be raised outside of the package.

b. Exceptions raised within a task should always be handled within that task. Note that in the case of an exception raised during a rendezvous, the exception will also be propagated back to the point of the entry call.

c. The predefined exceptions should generally not be explicitly raised.

3.6.1.2.9.4 Exception Propagation. Exceptions should not be allowed to propagate outside their own scope. An exception may be allowed to propagate to any point where it can be named in an exception handler. Note that this includes the case where an exception is defined in a package specification and has its scope "expanded" by a "with" clause. What must be avoided are cases such as the following:

```
...
  procedure RAISE_EXCEPTION is

    HIDDEN_EXCEPTION : exception;

  begin -- RAISE_EXCEPTION
    raise HIDDEN_EXCEPTION;
  end RAISE_EXCEPTION;

begin -- MAIN_PROGRAM
```

```
RAISE_EXCEPTION;
-- "HIDDEN EXCEPTION" CANNOT be named at this point
...
```

3.6.1.2.9.5 Suppressing Checks. Checks should not be suppressed except for essential efficiency or timing reasons in thoroughly tested program units.

3.6.1.2.10 Generic Units. A generic unit is a program unit that is either a generic subprogram or a generic package. A generic unit is a "template," which is parameterized or not, and from which corresponding (non-generic) subprograms or packages can be obtained. The resulting program units are said to be "instances" of the original generic unit.

3.6.1.2.10.1 Generic Formal Subprograms. The actual subprograms associated with the formal subprogram parameters of a generic unit should be consistent with the conceptual meanings of the formal parameters (e.g., only functions which are conceptually "adding operations" should be associated with a formal parameter named "plus").

Operator symbol function generic parameters should generally be provided with a box default body ("is <>").

```
with function "<" ( X, Y : ITEM ) return BOOLEAN is <>;
```

3.6.1.2.10.2 Use of Attributes. In writing generic bodies, attributes should be used as much as possible to generalize the code produced.

3.6.1.2.11 Representation Clauses and Implementation-Dependent Features. Usage of any features discussed in this paragraph requires the approval of the contractor Project Manager (final approval by the Government Project Officer).

3.6.1.2.11.1 Utilization. Machine dependent and low-level Ada features should not be used except when absolutely necessary. Representation clauses and implementation-dependent features should only be used for one of the following:

a. To increase efficiency (when absolutely necessary)

b. For interrupt handling

c. To specify task storage size

d. For address clauses to be used with entries only to associate them with hardware interrupts.

Representation clauses should not be used to change the meaning of a program.

3.6.1.2.11.2 Interrupts. Interrupt routines should be kept as short as possible. Usage of interrupts may not be approved by the contractor Project

Manager. Approval by the Government Project Officer is required prior to usage.

3.6.1.2.12 Input-Output. Usage of any features discussed in this paragraph requires the approval of the contractor Project Manager (final approval by the Government Project Officer).

3.6.1.2.12.1 Text Formatting. Line and page formatting should be done using the NEW_LINE and NEW_PAGE subprograms, rather than explicitly writing end-of-line or end-of-page characters.

3.6.1.2.12.2 Low-Level Input-Output. Use of package LOW_LEVEL_IO should be avoided unless absolutely necessary. Approval for usage of this feature must be obtained from the Government Project Officer (the contractor Project Manager may not approve usage of this Ada feature).

3.6.1.2.12.3 FORM Parameter. Use of the FORM parameter of the OPEN and CREATE procedures should generally be avoided. The "Form" parameter on the file OPEN and CREATE procedures specifies system-dependent file characteristics. This can reduce both readability and portability, and so should only be used if absolutely necessary.

3.6.1.3 Format Guidelines. The following subparagraphs specifically delineate the guidelines to be used in Ada source code generation. These guidelines are grouped into the following topics: lexical elements; declarations and types; names and expressions; statements; subprograms; packages; tasks; compilation units; exception declarations; generic units; and representation clauses.

3.6.1.3.1 Lexical Elements. The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate lexical elements. Each lexical element is either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character literal, a string literal, or a comment. Each lexical element must fit on one line, since the end of a line is a separator.

3.6.1.3.1.1 Indentation. The standard indentation is two spaces.

3.6.1.3.1.2 Character Set. Full use should be made of the International Standards Organization (ISO) character set where available. Alternate character replacements should only be used when the corresponding graphical symbols are not available.

3.6.1.3.1.3 Uppercase/Lowercase. Reserved words and attributes should appear in lowercase. Type, user-specified variable names, and enumeration value identifiers should appear in all uppercase.

```
LONG_INTEGER
AUTHORITY_LEVEL
(RED, GREEN, BLUE)
```

```
(ARMY, AIR_FORCE, NAVY, MARINES)
```

3.6.1.3.1.4 Identifiers. When specifying identifiers in source code, the following rules must be utilized:

a. Identifier names should be meaningful and easily distinguishable from each other, except possibly for loop parameters, array indices, and common mathematical variables, which may be as short as only one character.

b. Distinct words in identifiers should always be separated by underscores "_".

c. The use of abbreviations in identifiers should be avoided. When used, an abbreviation should be significantly shorter than the word it abbreviates, and its meaning should be clear. The same abbreviations should be used consistently throughout a project.

3.6.1.3.1.5 Spaces. Single spaces should be used consistently between lexical elements to enhance readability.

3.6.1.3.1.6 Blank Lines. Blank lines should be used to group logically related lines of text and a blank line should always follow a construct whose last line is not at the same indentation level as its first line.

```
type COMPLEX is
  record
     REAL      : FLOAT;
     IMAGINARY : FLOAT;
  end record;

-- Followed by a blank line
```

3.6.1.3.1.7 Continuations. Statements extending over multiple lines should always be broken BEFORE reserved words, operator symbols, or one of the following symbols:

```
:  |  =>  ..  :=
```

but they should be broken AFTER a comma (","). Unless otherwise specified in later guidelines, all the continuation lines should be indented at least two levels -- this is, four spaces -- with respect to the original lines they continue.

```
CORRECTED_VALUE := (1 + SENSOR_SCALE) * RAW_VALUE
    + DISTORTION_FACTOR * DISTORTION_VALUE + SENSOR_BIAS;
```

Long strings extending over more than one line should be broken up at natural boundaries, appropriate to the meaning of the contents of the string, if any.

```
"This is a rather long string, so it is likely that "
    & "it will extend over more than one line"
```

3.6.1.3.1.8 Comments. Comments should begin with the "--" aligned with the indentation level of the code that they describe, or to the right of the code, aligned with other such comments.

```
-- Check if the user has special authorization
if AUTHORITY = SPECIAL then
  DISPLAY_SPECIAL_MENU;        -- All operations are allowed
else
  DISPLAY_NORMAL_MENU;         -- Only normal operations allowed
end if;
```

3.6.1.3.2 Declarations and Types. Ada defines several kinds of entities that are declared, either explicitly or implicitly, by declarations. Certain forms of declaration always occur explicitly as part of a basic declaration. These forms are discriminant specifications, component declarations, entry declarations, parameter specifications, generic parameter specifications, and enumeration literal specifications. A loop parameter specification is a form of declaration that occurs only in certain forms of a loop statement. The remaining forms of declaration are implicit: the name of a block, the name of a loop, and a statement label.

A type declaration declares a type. The elaboration of a full type declaration consists of the elaboration of the discriminant part, if any (except in the case of the full type declaration for an incomplete or private type declaration), and of the elaboration of the type definition.

3.6.1.3.2.1 Commenting. Type declarations (or groups of declarations) should be commented to indicate what is being defined, if that is not obvious from the type declaration itself.

```
type VELOCITY is     -- Inertial velocity relative to the Earth
  array (1..3) of FLOAT;
```

Object declarations should be commented if the object definition is unclear from the object and type identifiers alone. Note that those properties of an object obtained from its type should not be repeated in comments on the object declaration.

```
SPACECRAFT_VELOCITY  -- Spacecraft orbital velocity, assuming a
  : VELOCITY;        -- circular orbit
```

3.6.1.3.2.2 Indentation. All declarations in a single declaration part should begin at the same indentation level.

3.6.1.3.2.3 Type Definitions. Array type definitions should have one of the following formats:

```
type <type name> is array <index definition> of <subtype indicator>;
---------------------------------------------------------------------
type <type name> is
  array <index definition> of <subtype indicator>;
---------------------------------------------------------------------
type <type name> is
  array <index definition>
    of <subtype indicator>;
```

Record type definitions should have one of the following formats:

```
type <type name> is
  record
    <component declaration>;
    <component declaration>;
  end record;
-----------------------------------
type <type name>
  ( <discriminant declaration>;
    <discriminant declaration> ) is
  record
    <component declaration>;
    case <discriminant name> is
     when <choices> =>
         <component declaration>;
         <component declaration>;
    end case;
  end record;
```

All <component declarations> and <discriminant declarations> should be formatted like object declarations (see subparagraph 3.6.1.3.2.4).

Other type definitions should be formatted as follows:

```
type <type name> is <type definition>;
-------------------------------------------
subtype <type name> is <subtype indicator>;
```

Long enumeration type declarations should be formatted into easily readable columns.

3.6.1.3.2.4 Object Declarations. Object declarations should have one of the following formats. The preferred formats are:

```
<object name> : <subtype indicator> := <expression>;
----------------------------------------------------
<object name> : <subtype indicator>
  := <expression>;
```

In the first case, all such declarations textually grouped together or appearing as components in a single record definition or in a single parameter list should have their ":" and ":=" symbols aligned.

3.6.1.3.3 Names and Expressions. Names denote declared entities, whether declared explicitly or implicitly. Names can also denote objects designated by access values; subcomponents and slices of objects and values; single entries, entry families, and entries in families of entries. Finally, names can denote attributes of any of the foregoing.

An expression is a formula that defines the computation of a value. The type of an expression depends only on the type of its constituents and on the operators applied. For an overloaded constituent or operator, the determination of the constituent type, or the identification of the appropriate operator, depends on the context.

3.6.1.3.3.1 Names. The name for a type should be a common noun indicating the class of the objects it contains. For example:

```
DEVICE
AUTHORITY_LEVEL
USER_NAME
PHONE_LIST
```

A type name may also end with the suffix "TYPE". For example:

```
EMPLOYEE_TYPE
SCHEDULE_TABLE_TYPE
COLOR_TYPE
```

The names of non-BOOLEAN valued objects should be nouns, preferably more precise than the names of types. For example:

```
CURRENT_USER   : USER_NAME;
OUTPUT_DEVICE  : DEVICE;
SCHEDULE_TABLE : SCHEDULE_TABLE_TYPE;
NEW_EMPLOYEE   : EMPLOYEE_TYPE;
```

BOOLEAN valued objects should have predicate-clause (e.g., "IS_OPEN") or adjective names. For example:

```
USER_IS_AVAILABLE
LIST_EMPTY
DONE
NOT_READY
IS_WAITING
```

3.6.1.3.3.2 Parentheses. Syntactically redundant parentheses should generally be used to enhance the readability of expressions, especially by indicating the order of evaluation. For example:

```
VARIANCE := (ROLL_ERROR ** 2) + ((YAW_ERROR ** 2) / 2);
```

3.6.1.3.3.3 Aggregates. When longer than two or three components, or whenever readability is improved, named aggregates should be formatted as indicated below, with one association per line and the "=>" arrows aligned.

```
OUTPUT_DEVICE :=
  (DEVICE          => DISK,
   STATUS          => CLOSED,
   CYLINDER        => 1,
   TRACK           => STARTUP_TRACK_NUM);
```

Aggregates for tabular data structures may instead be formatted in a tabular fashion, so as to enhance readability.

3.6.1.3.3.4 Continuation. When a long expression is broken over more than one line, it should be broken near the end of the line before an operator symbol with the lowest reasonable precedence.

```
CORRECTED_VALUE := (1 + SENSOR_SCALE) * RAW_VALUE
    + DISTORTION_FACTOR * DISTORTION_VALUE + SENSOR_BIAS;
```

3.6.1.3.4 Statements. A statement defines an action to be performed and is either simple or compound. A simple statement encloses no other statement. A compound statement can enclose simple statements and other compound statements.

3.6.1.3.4.1 Statement Sequences. Blank lines should be used liberally to break sequences of statements into short, meaningful groups (see subparagraph 3.6.1.3.1.6).

```
PUT_LINE ("Welcome to the Electronic Message System");


LOGON_USER (CURRENT_USER);
USER_DIRECTORY.LOOKUP
  (USER_NAME => CURRENT_USER,
   AUTHORITY => USER_AUTHORITY);

if USER_AUTHORITY = SPECIAL then
  PUT_LINE ("** You have SPECIAL authorization **");
end if;
```

3.6.1.3.4.2 IF Statements. Multiple conditions in an "if" clause should be grouped together, placed on appropriate lines, and aligned so as to enhance clarity. "If" statements should have the following format:

```
if <condition> then
  <statement>;
  <statement>;
```

```
elsif <condition> then
  <statement>;
  <statement>;
else
  <statement>;
  <statement>;
end if;
```

3.6.1.3.4.3 CASE Statements. "Case" statements should have the following format and note that the arrows "=>" should be aligned:

```
case <expression> is
  when <choices> =>
     <statement>;
     <statement>;
  when others    =>
     <statement>;
     <statement>;
end case;
```

3.6.1.3.4.4 LOOP Statements. A loop should preferably have a loop identifier (name) and must have one of the following formats:

```
<loop name>:
<iteration scheme> loop
  <statement>;
  <statement>;
end loop <loop name>;
-----------------------
<iteration scheme> loop
  <statement>;
  <statement>;
end loop;
```

3.6.1.3.4.5 Block Statement. Blocks should always have a block identifier (name) and should use the following format:

```
<block name>:
declare
  <declaration>;
  <declaration>;

begin -- <block name>
  <statement>;
  <statement>;

exception
  when <exception> =>
     <statement>;
     <statement>;
```

```
end <block name>;
```

3.6.1.3.5 Subprograms. A subprogram is a program unit whose execution is invoked by a subprogram call. There are two forms of subprograms: procedures and functions.

3.6.1.3.5.1 Subprogram Names. Except as indicated below, a subprogram name should be an imperative verb phrase describing its action:

```
OBTAIN_NEXT_TOKEN
INCREMENT_LINE_COUNTER
CREATE_NEW_GROUP
```

Non-BOOLEAN valued function names may also be noun phrases:

```
TOP_OF_STACK
X_COMPONENT
SUCCESSOR
SENSOR_READING
```

BOOLEAN valued functions should have predicate-clause names:

```
STACK_IS_EMPTY
LAST_ITEM
DEVICE_NOT_READY
```

File names must always accurately represent the program unit name to the extent allowable by the constraints of the operating system.

3.6.1.3.5.2 Subprogram Header. Each subprogram body or stub should be preceded by a header comment block containing the documenting information as described and shown in example form in figure 3-3.

3.6.1.3.5.3 Subprogram Declarations. Procedure declarations should have one of the following formats:

```
procedure <procedure name> (<parameter spec>);
----------------------------------------------
procedure <procedure name> (<parameter spec>;
                            <parameter spec>);
```

Each <parameter spec> should be formatted like an object declaration (see subparagraph 3.6.1.3.2.4).

Function declarations should have one of the following formats:

```
function <function name> (<parameter spec>) return <type mark>;
---------------------------------------------------------------
function <function name> (<parameter spec>;
                          <parameter spec>) return <type mark>;
```

```
-- ******************************************************************
-- * Unit Name  : SOME_PROCESS (SPEC, BODY, STUB, or SUBUNIT)          *
-- * Author     : Joe Analyst                                          *
-- * Create Date: 03/17/88                                             *
-- *                                                                   *
-- * Revision History:                                                 *
-- *   05/30/88 DH <Summarize exactly what changes were done            *
-- *               and why>.                                           *
-- *                                                                   *
-- * Purpose:                                                          *
-- *   Explain what this unit does to support the CSCI.                *
-- *                                                                   *
-- * Function:                                                         *
-- *   Explain how this unit achieves its purpose (e.g., algorithms,   *
-- *   structures, objects, types, tasks).                             *
-- *                                                                   *
-- * External Units Accessed:                                          *
-- *   TEXT_IO      : "with"ed in to allow alphanumeric I/O.           *
-- *   SOME_FUNCTION: called to perform calculations.                  *
-- *   GKS_LIB      : referred to for GKS-type I/O.                    *
-- *                                                                   *
-- * Exceptions:                                                       *
-- *   DATA_ERROR:       handles variable type inconsistencies.        *
-- *   CONSTRAINT_ERROR: handles range constraint errors.              *
-- *                                                                   *
-- * Hardware Dependencies:                                            *
-- *   List the access type, purpose, and justification for any,      *
-- *   such as: registers, memory, bulk storage, ports, and machine    *
-- *   code.                                                           *
-- *                                                                   *
-- * Compiler Dependencies:                                            *
-- *   List any special requirements and their justification, such     *
-- *   as: special pragmas, optional implementation, and specific      *
-- *   compilers.                                                      *
-- ******************************************************************
```

Figure 3-3. Ada Language Header Comment Block

Each <parameter spec> should be formatted like an object declaration (see subparagraph 3.6.1.3.2.4).

Parameter mode indications should always be used in procedure specifications. In a function specification, mode indications should either be used for all of the parameters or none.

3.6.1.3.5.4 Subprogram Bodies and Stubs. Subprogram bodies should have the following format:

```
separate (<parent name>)
<subprogram specification> is

-- <header comment block>
  <declaration>;
  <declaration>;

begin -- <subprogram name>
  <statement>;
  <statement>;
exception
  when <exceptions> =>
     <statement>;

end <subprogram name>;
```

The <subprogram specification> should be formatted as in a subprogram declaration (see subparagraph 3.6.1.3.5.3) and the <header comment block> is shown in figure 3-3.

Subprogram stubs should have the following format:

```
<subprogram specification> is separate;
```

where the <subprogram specification> is formatted as in a subprogram declaration (see subparagraph 3.6.1.3.5.3).

3.6.1.3.5.5 Named Parameter Association. Named parameter association should generally be used for procedure calls of more than a single parameter. Positional parameters are generally preferred for function calls. Named and positional parameter associations should not be mixed in a single subprogram call.

Named parameter associations should appear one to a line with formal parameters, "=>" symbols and actual parameters aligned. For example:

```
OBTAIN_NEXT_TOKEN
  (FILE      => CURRENT_SOURCE_FILE,
   POSITION  => CURRENT_COLUMN,
   TOKEN     => NEXT_TOKEN);
```

3.6.1.3.6 Packages. Packages allow the specification of groups of logically related entities. In their simplest form, packages specify pools of common object and type declarations. More generally, packages can be used to specify groups of related entities including subprograms that can be called from outside the package, while their inner workings remain concealed and protected from outside users.

3.6.1.3.6.1 Package Names. A package name should be a noun phrase describing the abstract entity modeled by the package, or simply whatever is being packaged:

```
STACK_HANDLER
VEHICLE_CONTROLLER
TERMINAL_OPERATIONS
PARSER_TYPES
UTILITIES_PACKAGE
```

File names must always accurately represent the program unit name to the extent allowable by the constraints of the operating system.

3.6.1.3.6.2 Package Header. Each package specification, body, or stub should be preceded by a header comment block as shown in figure 3-3.

3.6.1.3.6.3 Package Specifications. Package specifications should have the following format:

```
package <package name> is

-- <header comment block>
  <declaration>;
  <declaration>;

private
  <declaration>;
  <declaration>;

end <package name>;
```

The <header comment block> is shown in figure 3-3.

In a declarative part, all package specifications should appear before any package or task bodies.

3.6.1.3.6.4 Package Bodies and Stubs. Package bodies should have the following format:

```
separate (<parent name>)
package body <package name> is

-- <header comment block>
```

```
    <declaration>;
    <declaration>;

  begin -- <package name>
    <statement>;
    <statement>;
  exception
    when <exceptions> =>
      <statement>;

  end <package name>;
```

The <header comment block> is shown in figure 3-3.

Package stubs should have the following format:

```
  package body <package name> is separate;
```

3.6.1.3.7 <u>Tasks</u>. Tasks are entities in Ada whose executions proceed in parallel. Each task can be considered to be executed by a logical processor of its own. Different tasks proceed independently, except at points where they synchronize.

3.6.1.3.7.1 <u>Task and Entry Names</u>. A task name should be a noun phrase describing the task function or abstract entity modeled by the task. For example:

```
  SENSOR_INTERFACE
  STATUS_MONITOR
  EVENT_HANDLER
  MESSAGE_BUFFER
```

Entry names should follow the same guidelines as for procedure names (see subparagraph 3.6.1.3.5.3).

3.6.1.3.7.2 <u>Task and Entry Headers</u>. Each task or task type specification or body and each entry specification should be preceded by a header comment block as shown in figure 3-3.

3.6.1.3.7.3 <u>Task Specifications</u>. Task specifications should have the following format:

```
  task <task name> is

  -- <header comment block>
    <declaration>;
    <declaration>;

  end <task name>;
```

The <header comment block> is shown in figure 3-3 and a task type specification should be formatted the same as a task specification.

Entry declarations should have the following format:

```
entry <entry name> (<family range>)
  (<parameter specification>;
   <parameter specification>);

-- <header comment block>
```

Each <parameter specification> should be formatted like an object declaration (see subparagraph 3.6.1.3.2.4). The <header comment block> is shown in figure 3-3.

Parameter mode indications should always be used in entry declarations and in a declarative part, all task specifications should appear before any task or package bodies.

3.6.1.3.7.4 <u>Task Bodies and Stubs</u>. Task bodies should have the following format:

```
separate (<parent>)
task body <task name> is

-- <header comment block>
  <declaration>;
  <declaration>;

begin -- <task name>
  <statement>;
  <statement>;

exception
  when <exceptions> =>
     <statement>;

end <task name>;
```

The <header comment block> is shown in figure 3-3.

Task stubs should have the following format:

```
task body <task name> is separate;
```

3.6.1.3.7.5 <u>ACCEPT Statements</u>. Parameter mode indications should always be used in "Accept" statements and they should have one of the following formats:

```
accept <entry name> (<entry index>);
------------------------------------
```

```
accept <entry name> (<entry index>)
  (<parameter specification>;
   <parameter specification>);
do
  <statement>;
  <statement>;
end <entry name>;
```

Each <parameter specification> should be formatted like an object declaration (see subparagraph 3.6.1.3.2.4).

3.6.1.3.7.6 SELECT Statements. Selective wait statements should have the following format:

```
select
  <statement>;
  <statement>;
or
  <statement>;
  <statement>;
or
  when <condition> =>
    <statement>;
    <statement>;
else
  <statement>;
  <statement>;
end select;
```

This format is consistent with the indentation style of other statements. In addition, the added level of indentation especially highlights guarded sections of code.

Conditional and timed entry calls should have the following format:

```
select
  <entry call>;
  <statement>;
else
  <statement>;
  <statement>;
end select;
```

3.6.1.3.7.7 Pragma Priority. The priority pragma should appear in task specifications before any entry declarations, and in the main program before any declarations.

3.6.1.3.8 Compilation Units. Each compilation unit should be in a separate file, except possibly in the case of a generic procedure specification and its body.

3.6.1.3.9 Exception Declarations. Exception declarations should be formatted like object declarations (see subparagraph 3.6.1.3.2.4).

3.6.1.3.10 Generic Units. A generic unit is a program unit that is either a generic subprogram or a generic package. A generic unit is a "template," which is parameterized or not, and from which corresponding (non-generic) subprograms or packages can be obtained. The resulting program units are said to be "instances" of the original generic unit.

3.6.1.3.10.1 Generic Declarations. Generic declarations should have the following format:

```
generic
  <declaration>;
  <declaration>;
<program unit specification>;
```

Each <declaration> should be formatted like its non-formal counterpart (see subparagraphs 3.6.1.3.2.3 and 3.6.1.3.2.4), except for formal subprograms which should be formatted as a generic formal parameter subprogram shown below. The <program unit specification> should be formatted as for non-generic units (see subparagraphs 3.6.1.3.5.3 and 3.6.1.3.6.3).

A generic formal parameter subprogram declaration should have one of the following formats:

```
with <subprogram specification>;
---------------------------------------------------
with <subprogram specification> is <>;
---------------------------------------------------
with <subprogram specification> is <default name>;
```

The <subprogram specification> should be formatted as for a subprogram declaration (see subparagraph 3.6.1.3.5.3).

A generic declaration should be preceded by the appropriate unit header block (see subparagraphs 3.6.1.3.5.2 and 3.6.1.3.6.2).

3.6.1.3.10.2 Generic Instantiations. Generic instantiations should have one of the following formats:

```
<unit header> is new <generic name> (<generic arg>,<generic arg>);
----------------------------------------------------------------- ----
<unit header> is new <generic name> (<generic parm => <generic arg>,
                                     <generic parm => <generic arg>);
```

Note that in the second form, the arrows ("=>") should be kept aligned.

3.6.1.3.11 Representation Clauses. Representation clauses should be placed near to the objects they affect.

3.6.2 C Style Specifications. The following subparagraphs present the structure, coding, and format guidelines to be utilized in C language development for all projects comprising the JNGG Graphics Program.

3.6.2.1 Structure Guidelines. The following subparagraphs present and discuss general guidelines to be used in the structure utilized for C language programming.

3.6.2.1.1 Functional Cohesion. A function or macro function should perform a single, conceptual action.

3.6.2.1.2 File Utilization. Files allow the specification of groups of logically related entities. In their simplest form, files specify pools of common object and type declarations. More generally, files can be used to specify groups of related entities including functions that can be called from outside the file, while their inner workings remain concealed and protected from outside users.

In order for a file to be truly useful, it must perform one or more of the following purposes which are listed below in order of decreasing desirability:

a. Model an abstract entity (or data type) appropriate to the domain of a problem. This is the strongest use of files for structuring a program. It corresponds to the requirement of functional cohesion (see subparagraph 3.6.2.1.1) and contributes to the goal of making the structure of a program reflect the structure of its problem domain.

b. Collect related type and object declarations which are used together (this kind of file should be used only to provide a common set of declarations for two or more library units). Further, it is better to minimize the declaration of variables in these files. Overuse of files of variables results in a FORTRAN COMMON block style program decomposition which defeats the abstraction and information hiding properties of files (see subparagraph 3.6.2.2.6.2).

c. Group together program units for essential configuration control or visibility reasons. This type of file should be used sparingly since it gives no additional information to a human reader on the structure of the program but might, for example, be used to divide a large program at the top level into subsystems to be developed by separate teams. However, it would be best if these subsystem files fulfilled at least one of the other two purposes in addition to this one.

Files should NOT be designed based on the procedural structure of the code which calls them. For example, a group of functions should not be packaged simply because they are all called at system initialization, or because they are always called in a certain sequence. Such a file is closely coupled to the context in which it is used and is not very understandable, reusable, or maintainable as a unit.

A logical hierarchy of files should be used to reflect or model levels of abstraction.

3.6.2.1.3 Scope of Visibility. Structural guidelines for the scope of identifiers should not extend further than necessary. This promotes information hiding and reduces coupling. The C language has four basic scope rules to bear in mind:

a. **External variables**: for variables declared outside a function, the scope is from the end of the declaration to the end of the file. External declarations must be placed at the front of the file.

b. **Local variables**: for variables declared **inside a function**, the scope is the rest of the block--from the end of the declaration to the end of the block. This limited scope of local variables means that a programmer does not need to worry about inadvertently using a name that has been used inside some other function.

c. **Visibility to the linker**: external variables declared as **static** will not be published to the linker; they will be known only within their source file. Other external variables will be published to the linker and linked together across source files. For an external variable to be known in a source file other than the one containing its **definition**, the second source file must contain an **extern** declaration for the variable.

d. **Function names**: the names of functions must explicity note whether or not the function is to be **external** or **static**. Thus, it is mandatory that the keywords **extern** or **static** be added to function declarations.

3.6.2.1.4 Program Structure and Compilation Issues. The overall structure of programs and the compilation issues relevant to style guidelines are described in this subparagraph.

3.6.2.1.4.1 Program Units. The compilation units of a program belong to a "program library." The following structural guidelines apply:

a. Library units should be used to allow configuration control of the high-level functional subsystems of a program and for reusable program units.

b. Library units in a file structure are preferable to library units which are functions. Library units providing services to the main program should always be files.

3.6.2.1.4.2 INCLUDE Clauses. A context clause is used to specify the library units whose names are needed within a compilation unit. No unit should have an "include" clause for a unit it does not need to see directly.

3.6.2.1.4.3 Program Unit Dependencies. The following guidelines pertinent to program unit dependencies should be adhered to:

a. Excessive dependencies between compilation units should be avoided.

b. It is preferable to limit program unit dependencies to a tree structure whenever possible.

3.6.2.1.5 Implementation-Dependent Features. Implementation-dependent features should, if possible, be hidden inside files which present implementation-independent interfaces to programmers.

3.6.2.1.6 Use of Prototypes. It is required that each C function in a file have a corresponding prototype function declaration in its accompanying header file.

3.6.2.2 Coding Guidelines. The following subparagraphs define and discuss guidelines to be used in C source code generation. These guidelines are grouped into the following topics: declarations and types, preprocessing guidelines, guidelines for scalars, arrays, pointers, structures, dynamic storage allocation, and opening named files.

3.6.2.2.1 Declarations and Types. The following subparagraphs provide coding guidelines relevant to object declarations, constants, enumeration types, and floating-point types.

3.6.2.2.1.1 Constants. C language constants must utilize the **#define** construct to make them easier to read and maintain. Thus, a declared object is a constant if named through use of **#define** and an explicit initialization is provided.

An object should be declared constant if its value is intended not to change. Declaring an object to be constant clearly signals both the human reader and the compiler the intention that its value will not change. This not only increases readability, it also increases reliability.

```
#define LINES_PER_PAGE 66            /* integer constant */
#define PI 3.14159                   /* numeric constant */
#define LC_A 'a'                     /* character constant */
#define MESSAGE "Improper input\n"   /* string constant */
```

3.6.2.2.1.2 Enumeration Types. Recent C compilers provide enumeration types, which allow declarations like this:

```
enum stopLight {red, yellow, green}
```

This establishes **enum stopLight** as a type which can be used in further declarations, such as:

```
enum stopLight mainStreet, frontStreet;
```

And the enumeration constants (red, yellow, and green) become symbolic names for the integer constants 0, 1, and 2. Each of the enumeration constants can be explicitly initialized, as in:

```
enum stopLight {red = 'r', yellow = 'y', green = 'g'};
```

An enumeration variable is treated just like an **int** variable, except that its actual storage size might be optimized according to the range of the enumeration constants.

3.6.2.2.1.3 Floating-Point Types. The exact form of the storage of floating-point numbers varies with the machine, but all standard C compilers offer two types of floating-point numbers:

float which holds at least 6 decimal digits; and

double which holds at least 15 decimal digits.

Since machines differ in their floating-point mechanisms, answers may be slightly different on different machines. In any case, on any machine, answers are only approximate--accurate only to a limited number of decimal places. Comparing two floating-point results for exact equality is generally risky. Adding a long series of numbers can create a noticeable round-off error; the more numbers in the series, the larger the error. Subtracting two nearly equal numbers can also create a round-off error. A small round-off error can also creep in when decimal input is converted to internal binary format or vice versa.

These considerations suggest two cautionary notes. First, before attempting serious scientific or engineering computations with long sequences of floating-point operations, first consult a text on numerical analysis. Second, if floating-point numbers are used for dollars and cents computations, be aware that a printed result could be off by a penny.

A number of useful functions are available in the C library for floating-point computation. Some of the more common functions are:

ceil(x)	smallest integer not less than x
cos(x)	cosine of x
exp(x)	exponential function of x
floor(x)	largest integer not greater than x
log(x)	natural log of x
log10(x)	base-10 log of x
pow(x, y)	raise x to the power y
sin(x)	sine of x
sqrt(x)	square root of x
tan(x)	tangent of x

3.6.2.2.1.4 <u>Object Declarations</u>. Each object declaration should declare only one object even though the objects may be of the same type. Where possible, objects should always be initialized by their declaration, rather than in later code. This practice enhances the readability and reliability of the code:

```
int  keyStroke     = 25;
int  numberItems   = 2;
char messageCode[7] = "DM0745";
```

3.6.2.2.2 <u>Preprocessing Guidelines</u>. The "definition" capabilities of C provide techniques for writing programs that are more portable, more readable, and easier to modify reliably. This subparagraph presents a number of guidelines related to definitions and header files that aid in building reliable C programs.

3.6.2.2.2.1 <u>Defined Constants</u>. The following subparagraphs discuss various guidelines and rules to be considered when using defined constants--both those provided by standard C header files and those defined during software development.

3.6.2.2.2.1.1 <u>Definitions Containing Operators</u>. Any macro definition containing operators needs parentheses around the entire definition. Each appearance of a macro argument in the definition also needs to be parenthesized if an embedded operator in the argument could cause a precedence problem. For example:

```
#define EOF (-1)     /* Good */
#define EOF -1       /* Bad  */
```

The first definition causes any subsequent appearance of the name EOF to be replaced by the characters (-1). Since the definition contains an operator (the "minus"), it is enclosed in parentheses. The second definition, however, could possibly generate a syntactically correct, but unintended result. If a programmer erroneously coded:

```
if (c EOF)     /* Should be (c != EOF) */
```

The compiler would interpret it as:

```
if (c - 1)     /* Syntactically correct, but unintended */
```

3.6.2.2.2.1.2 <u>Need for Environmental Capability</u>. Reliable modification of defined constants requires an environmental capability. There must be a means for ensuring that all files comprising a program have been compiled using the same set of headers (e.g., the Unix MAKE command).

Although readability is the main reason for creating defined constants, modifiability is a close second. The name BUFFER_SIZE tells the reader that it is the "buffer size" used in efficient I/O transfers (e.g. #define BUFFER_SIZE 512). However, the defined constant also shows how the program can be modified for a different value. Thus, on systems where 1024 is a better value for the size of disk I/O transfer, the standard header STDIO.H could specify the value 1024 for **BUFFER_SIZE**.

3.6.2.2.2.1.3 Commenting Modifiability Limitations. If there are limitations on the modifiability of a defined constant, indicate the limitations with a comment:

```
#define EOF (-1)    /* DO NOT MODIFY: ctype.h expects -1 value */
```

Another application of this guideline is the explicit indication of minimum and maximum values:

```
#define NBUFS 5   /* min 2, max 30 */
```

3.6.2.2.2.1.4 Relationships Between Definitions. If one definition affects another, embody the relationship in the definition. Do not use two separate definitions. For example:

```
#define VALUE   5
#define VALUE_2 (VALU + 2)
```

The previous pair of definitions follows the rule by showing the relationship, whereas:

```
#define VALUE   5
#define VALUE_2 7  /* MISLEADING -- no indication of relationship */
```

does not. In the former case, the program could be modified reliably by changing the definition of VALUE. In the latter example, a guess would be required to figure out that VALUE_2 should probably be changed to equal VALUE plus two.

3.6.2.2.2.1.5 Use of Expressions. If a value is given for a defined constant, don't defeat its modifiability by assuming its value in expressions. Just giving the constant a name is not enough to ensure modifiability.

The programmer must always use the name and remain cognizant of the possibility that the value could change. For example:

```
#define BUFFER_SIZE 512
.
.
.
nblocks = nbytes >> 9;   /* Used in multiple places in the program */
```

Here, the programmer makes the assumption that "everyone knows" that BUFFER_SIZE equals 512, and right-shifting 9 bits is the same (for positive numbers) as dividing by 512. However, if BUFFER_SIZE changes to 1024 on another system, modifications to the buffer size used throughout the program would be rather difficult. Instead, the programmer should have used:

```
nblocks = nbytes / BUFFER_SIZE;  /* Easily maintainable expression */
```

3.6.2.2.2.1.6 Use of Standardized Environment-Dependent Limits. Use LIMITS.H for environment-dependent values. In this manner, system dependencies are limited to the appropriate header file and the numbers of previously required definitions in the source code is much less.

3.6.2.2.2.2 Defined Types. Use a consistent set of project-wide defined types. For the purpose of engineering reliable software, it is crucial that more precise definitions of data types (than those basic types supplied by C) are made. Special semantic rules apply for some of these defined types and they will be described below. In particular, there are five defined types that would be useful to most applications:

```
bits      /* an unsigned short integer used for bitwise operations */
ushort    /* an unsigned short integer used for arithmetic */
metachar  /* a short integer holding a char value or EOF */
bool      /* an integer to be tested for zero or non-zero */
void      /* the "return type" for a function that returns no value */
```

These types may be defined using the **typedef** statement. An example appears below:

```
typedef unsigned short ushort, bits;
typedef short metachar;
typedef int bool;
typedef int void;  /* delete if compiler supports void */
```

Although **#define** could also be used for this purpose, an important difference between **#define** and **typedef** is that **#define** replaces the name with its definition during preprocessing, whereas **typedef** is handled by the C syntax analysis. Thus, a defined types created via **typedef** cannot be "undefined" or re-defined, so its usage is more reliable. Also, if the name is mistakenly used as a variable name, the diagnostics are more intelligible.

In terms of portability, the symbols **ushort** and **void** are important. Not all compilers currently accept the **unsigned short** type, but most that do not accept it are targeted for small machines where **ushort** can simply be translated into **unsigned int**.

Regarding **void**, any function that does not return a value should be indicated as being a **void** function. Recent C compilers implement **void** as a keyword. If the compiler used for the project supports **void**, do not try to **typedef** a definition for it. Instead, remove any such definitions from the appropriate header files. If the compiler used does not support void, it should be defined as **int**.

3.6.2.2.2.3 Standard Headers. Headers are used for several purposes: creating defined constants, creating defined types, and creating prototypes. Separate headers should be created for each related group of functions, along with any special symbols that are useful with those functions.

3.6.2.2.2.3.1 Function Declarations. Declare all functions in header files before they are used. The returned type of each function is thus declared so as to eliminate any possible problems with the compiler misinterpreting the function's returned type. The types of the function parameters should also be specified in comments.

3.6.2.2.2.3.2 Local Headers. Create a portability definitions header (e.g., portdefs.h) local to the project. It should contain various defined types, defined constants, and macros that will be important in producing portable code. This header should be included in all programs to ensure that all project programs are compiled with "portability definitions."

3.6.2.2.2.4 Macro Functions. A macro with parameters will be referred to as a macro function. Macro functions may be used only with contractor Project Manager approval (final approval given by the Government Project Officer). An example of a macro function is:

```
#define ABS(x) (((x) < 0) ? -(x) : (x))
```

Since each argument can contain operators, each parameter is parenthesized to avoid precedence conflicts. Furthermore, since the entire result is an expression usable with other operators, the entire definition is also parenthesized (refer to subparagraph 3.6.2.2.2.1.1).

There are two types of macro functions: safe and unsafe. A **safe** macro function evaluates each parameter only once in the code expansion, whereas an unsafe macro function is one which evaluates a parameter more than once in the code expansion. By this definition, the preceding example is an **unsafe** macro function.

Thus, a critical issue of concern when writing macro functions are the side-effects on macro arguments. The following example would increment <n> twice:

```
ABS(++n)  /* BUG! */
```

Therefore, the documentation for such macros must warn about putting side-effects on the invocation, and the responsibility is upon the programmer using the macro (refer to subparagraph 3.6.2.2.2.1.3).

3.6.2.2.2.4.1 Naming of Unsafe Macros. Use uppercase names for unsafe macro functions to emphasize the restrictions on their usage. For safe macros, there are some advantages to using lowercase names. Each safe macro could be replaced by an actual function call, and at different times during project development, one might want the macro version or the function version.

3.6.2.2.2.4.2 Invoking Unsafe Macros. Never invoke an unsafe macro function with arguments containing assignment operations, increment/decrement operations, or function calls.

3.6.2.2.2.4.3 Safe Macro Usage. Use safe macro functions whenever possible. The usage of unsafe macro functions requires completion of an Approval Form and the approval of Government management.

3.6.2.2.2.5 Undefining. In general, the redefinition of symbols is quite unreliable. Each symbol should have an invariant meaning, so that each instance of the symbol denotes the same thing.

Usage of the **#undef** statement requires approval by the contractor Project Manager (final approval by the Government Project Officer).

3.6.2.2.2.6 Conditional Compilation. The preprocessor provides for conditional compilation, whereby some lines of code may be selectively excluded from compilation depending upon the outcome of some test using some combination of appropriate commands (e.g., #if, #else, #ifdef, #ifndef, #elif, and #if defined).

3.6.2.2.2.6.1 Commenting-Out Code. Sometimes one needs to "comment out" several lines of code. Most compilers (and ANSI C) do not test comments, so ordinary comments cannot be used. The following construct will delete the enclosed lines even if they contain other **#if** constructs:

```
#if 0
/*  ...code to be commented out... */
#endif
```

3.6.2.2.2.6.2 Usage of an Inclusion Sandwich. All header files will be enclosed in an "inclusion sandwich." Essentially, each header **#define**'s a symbol that means "I have already been included." In this way, the first time a header file is **#include**'d, all of its contents will be included. Subsequent attempts to **#include** the file will be ignored. The appropriate structure to be used is:

```
#ifndef HEADER_H
#define HEADER_H
/*  ...contents of the header...  */
#endif
```

3.6.2.2.3 Guidelines for Scalars. This subparagraph addresses the reliable use of scalars--an object having floating, integer, or pointer type.

3.6.2.2.3.1 The Math Library. The following subparagraphs provide guidelines and reliability concepts to be considered when using the C-provided math library--<math.h>.

3.6.2.2.3.1.1 Floating-Point to Integer Conversions. When a positive floating-point value is converted to an integer value, the fractional digits are truncated. However, when a negative floating-point value is converted, the truncation may be toward or away from zero, depending on the implementation.

Therefore, when exactness counts in converting floating-point to integer, the value being converted must be non-negative.

3.6.2.2.3.1.2 Testing for Errors. When testing for errors being returned, the global variable **errno** should be used. This variable is set to a non-zero value by any of the math functions that encounters an error. Thus, setting **errno** to zero prior to a computation, and testing it afterwards, will reveal whether any library errors were reported during the computation·

```
errno = 0;
<perform computation>
if (errno != 0)
  <handle the error situation: report, compute differently, etc.>
else
  <computation was error-free>
```

3.6.2.2.3.2 Character Tests. The facilities of the header file <ctype.h> must be used for character tests and uppercase and lowercase conversions. These functions are the most portable way of testing and converting characters.

3.6.2.2.3.3 Boolean Data. All test conditions must be clearly "Boolean" expressions. Any expression whose top-level operators are relational and logical is obviously a "Boolean" expression, but what about Boolean variables? From the compiler's point of view, any value other than zero or one might be assumed to be non-Boolean.

From the human reader's point of view, the program will be easier to understand if Boolean variables are clearly indicated as such. The following type scheme provides a defined type for Boolean variables:

bool   designates an **int**-sized Boolean variable

Therefore, always ensure that Boolean variables are assigned the values zero and one. This means that the type **bool** is always adequate.

Also, ensure that each test condition is Boolean, involving only the Boolean type or relational and logical operators.

3.6.2.2.3.4 Enumeration Types. If an un-initialized enumeration constant follows one which is initialized, its value is one greater than the previous constant. Therefore, an enumeration's constants must all be initialized, or else none of them should be initialized.

Furthermore, although enumeration variables are treated much like **int** variables, write programs as if enumeration variables could receive no values other than the associated enumeration constants. Treat the enumeration types as if they were unique types, not for any arithmetic **int** usages. Convert between enumeration variables and integer values only by use of an explicit cast.

3.6.2.2.3.5 Range-Checking. Failure to attend to proper ranges of variables can lead to interesting reliability problems. The following subparagraphs describe methods to be used in developed code to assure proper ranges are checked.

3.6.2.2.3.5.1 Modifying Loop Control Variables. The value of loop control variables may not be modified by a program.

3.6.2.2.3.5.2 Inclusion of "One-Too-Far" Values. In C language programs developed for the JNGG Graphics Program, the "one-too-far" value must be considered as part of the range of a variable if they are needed for loop terminations or other testing purposes.

For example, consider a variable named **monthNo** which is a subscript into an array of monthly information. Assume further that its range is 0 to 11. A common usage of such a variable appears below:

```
for (monthNo = 0; monthNo < 12; ++monthNo)
  <some_process> array[monthNo];
```

The **for** loop always involves a "one-too-far" value at the end of the loop; monthNo must always be incremented to 12 in order for the loop test (**monthNo < 12**) to terminate the loop.

3.6.2.2.3.5.3 Size_T Type Usage. The discussion of ranges would not be complete without considering the sizes of objects and the ranges of subscripts. C is moving toward the ability to handle objects whose size cannot be represented in an **unsigned int**. ANSI C envisions that some environments will need to use **unsigned long** to represent the **sizeof** anything. Several of the standard headers will define a type named **size_t**, which will be either **unsigned int** or **unsigned long**, according to the environment.

Therefore, function parameters accepting the size of an arbitrarily large object must be declared with the **size_t** type. The following example usage will clarify the usage of **size_t**:

```
reverse:
  /* reverse - reverse the order of a string */
  #include "local.h"
  void reverse(s)
    char s[];
    {
    char   t;
    size_t i, j;

    if ((j = strlen(s)) == 0)
      return;
    for (i = 0, j = j - 1; i < j; ++i, --j)
      SWAP(s[i], s[j], t);
    }
```

3.6.2.2.3.6 Signed and Unsigned Arithmetic. Signed and unsigned arithmetic has significantly different properties in C. The following subparagraphs discuss how best to support portability and maintainability in C code when utilizing such operations.

3.6.2.2.3.6.1 Subtraction Between Unsigned Integers. When two unsigned integers are subtracted, convert the result using either **(unsigned)** or UI_TO_I. Ensuring that this is done in our code will clarify the possible duality of interpretation that such operations will normally exhibit.

3.6.2.2.3.6.2 Usage of the Integer Modulo Macro (IMOD). The sign of the remainder operator (%) is implementation-dependent when the operands are of different signs. This will usually cause a portability problem when the programmer has assumed that **i % j** is always positive.

To provide a true (never negative) modulo operation, an IMOD ("integer modulo") macro from the portdefs.h header file should be used:

```
/* modulo macro giving non-negative result */
#define IMOD(i, j) (((i) % (j)) < 0 ? ((i) % (j)) + (j) : ((i) % (j)))
/* if i % j is never negative, replace with the following line: */
/* #define IMOD(i, j) ((i) % (j)) */
```

3.6.2.2.3.7 Overflow. In signed integer arithmetic, always assume that overflow is illegal, may be detected (hence should never be programmed), and cannot be trapped or ignored. In doing so, the C code will not be subject to intermediate overflow problems--an operation that produces a result that is too large for the intermediate result.

3.6.2.2.3.8 Data Properties. The most basic distinction when talking about data properties is the distinction between an **undefined** value (e.g., "garbage," or un-initialized), and a **defined** value. The following example shows an "undefined" error arising from forgetting to initialize a counter:

```
long sum;
int i;

for (i = 0; i < N; ++i)
  sum += a[i];             error - sum not initialized
```

In C, automatic variables are undefined until they are initialized and static variables are initialized by default, so they are initially defined. Thus, a variable can acquire an undefined value as the result of an invalid operation:

```
n = m / 0;
```

causes <n> to become undefined, possibly terminating execution as well.

Therefore, the importance of documenting the defining properties of declared names in a comment on the declaration becomes clear. Besides stating desired properties on the declaration of variables, the programmer can also document properties of function-returned values:

```
bool isEmpty();      /* is node empty?: bool */
```

All declared names must follow the above convention which provides a comment with a colon followed by a property name meaning "*must have this property.*"

3.6.2.2.4 Arrays. In C, arrays are used for storing character strings as well as for their more universal uses. As a result, the assurance of reliability for the properties of an array become more complex than in languages where strings are objects in their own right.

3.6.2.2.4.1 Array Data. The following subparagraphs describe the reliability concerns pertinent to the properties of array data as well as the guidelines that will help ensure that programming pitfalls are avoided.

3.6.2.2.4.1.1 Storage Class Precedence. In C, the declaration of one variable actually consists of five components:

```
static char line[10] = "msg";
```

where:	static	storage class (optional)
	char	type specifier
	line[1C]	declarator
	= "msg"	initialization (optional)
	;	semicolon (statement delimiter)

Strictly speaking, the storage class and the type specifier can appear in any order, but declarations are clearer in a conventional order. Thus, storage class (if any) should always precede the type specifier.

3.6.2.2.4.1.2 Optional Initialization of Variables. If a variable has an initialization, its declaration must have a source line to itself. This is due to the fact that several variables can be declared in a single declaration:

```
static char lineA[10], lineB[10];
```

The previous line is equivalent in every way to two separate declarations:

```
static char lineA[10];
static char lineB[10];
```

A common error is often made where the programmer assumes that the following initialization applies to both **lineA** and **lineB**:

```
static char lineA[10], lineB[10] = "msg";    <-- misleading
```

3.6.2.2.4.1.3 Array Properties. Document the defining property of a data object with a comment on its declaration. Ensure that this defining property remains invariant (unchanging) as much as possible throughout the computation, and document any exceptions.

An array is either **complete** (all scalar elements contained within the array are defined) or **incomplete** (one or more elements are not defined). An array can become complete either by assignment to all its elements, or by an initializer on its declaration. Thus, all arrays should be made complete before the array is used in order to make the program easier to write correctly and to understand.

Some arrays have other properties that are more important than "complete" or "incomplete." A **string**, for example, requires defined characters only up to a nul (i.e., '/0') terminator. The characters after the nul terminator can be total garbage and the array still has its defining property satisfied--being a string. Therefore, if an array's defining property can be true even if not all elements are defined, indicate that property on the array's declaration, thus:

```
char s[10];    /* : string */
```

Without this indication, it is assumed that the array must be complete (all elements defined) before its value is used.

3.6.2.2.4.2 Sorting an Array. When sorting arrays, use executable assertions whenever they are simpler than the code being protected and when the time to execute the assertions is not much greater than the time required to execute the code. They are much more reliable than simple comments indicating the intended invariant conditions of the sort process.

3.6.2.2.5 Pointers. Pointers are both powerful and dangerous in C. Programs can perform arbitrary manipulations of their data space in C and that aspect is essential for many of the system-level uses of C. However, a style of programming oriented toward portability and reliability must place restrictions on the uses of pointers.

3.6.2.2.5.1 Declaration of Pointers. In each pointer assignment, the right-hand side value must have exactly the same ("converted") pointer type as the left-hand side. In the assignment:

```
char s[10], *p;

p = s;
```

the "declared" type of **s** is **char [10]** and the "converted" type is **char ***, so the assignment is proper. By contrast, here is an improper assignment:

```
char s[10];
int *pi;

pi = s;       <-- improper mixed-type pointer assignment
```

3.6.2.2.5.2 Pointers to Scalars. A NULL pointer is one which contains the unique "null" value. A pointer becomes NULL when the integer zero is assigned to it. A NULL pointer always compares equal to integer zero. An **undefined** pointer is one that is not NULL, and yet is not pointing to an object of a compatible type (e.g., "garbage," un-initialized). Thus a **defined** pointer is either pointing to valid storage, or else is NULL. Thus, a pointer that is non-NULL is implied to be defined and points to valid storage. The importance of these concepts can be illustrated by considering various errors in calling **ShortOrder**.

The following example shows an undefined pointer error:

```
short *ps1, *ps2;

ShortOrder(ps1, ps2);
```

Without initializations or assignments, the values of **ps1** and **ps2** are undefined. In effect, **ShortOrder** has been asked to rearrange the contents of two random memory locations!

The following example shows a NULL pointer error:

```
short *ps1 = NULL;
short *ps2 = NULL;
```

```
ShortOrder(ps1, ps2);
```

Variables local to **ShortOrder** would receive the NULL value from the calling function and, since NULL does not point to any valid object, most compilers will attempt to access memory location zero, producing unpredictable results.

Therefore, the default requirement for pointer parameters is that they must point to storage that is entirely defined. Whenever a pointer parameter can accept something else, this should be explicitly stated on that parameter's declaration comment. This rule may or may not apply to a given project since the Unix operating system normally enforces this anyway.

3.6.2.2.5.3 Dangling Pointers. A pointer can become undefined if the object that it is pointing to should "disappear" during the lifetime of the pointer. An example appears below:

```
dangling.c:
  /* dangling - example of dangling pointer */

  #include "local.h"
  static short *pi = NULL;
  main()
    {
    void f1();

                   /* pi => NULL initially */
    f1();          /* pi => undefined suddenly! */
    }
  void f1()
    {
    short i;

    pi = &i;       /* pi => complete, momentarily */
    }
```

The return from function **f1** above causes the pointer **pi** to become "undefined." In other words, a dangling pointer.

Therefore, a function in which the address of an automatic variable is assigned to a non-automatic pointer must contain a comment to that effect. In any function with such a comment, each return from the function is an event requiring verification that no dangling pointers are left.

3.6.2.2.6 Structures. Structures are used for grouping a collection of objects into a single aggregate object. They can also be subdivided into the level of machine bits, using bit fields. Unions are syntactically very similar to structures, allowing the same memory to be used in different ways.

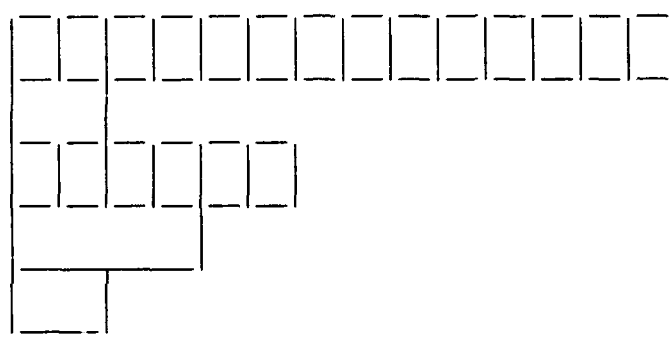
3.6.2.2.6.1 Records. The record structure is one of the oldest uses of a structure in programming. The following example collects information about a specific part in an assembly:

```
struct part
  {
  char  partNo[14];      /* part number: string */
  short leadTime;        /* lead time (in weeks): {0:99} */
  char  unitMeas[6];     /* unit of measure: string {"each","lb","box"} */
  float unitCost;        /* cost for one unit_meas: {0.00:9999.99} */
  short costQty;         /* quantity required for price: {0:9999} */
  };
```

In a typical environment where **short** requires two bytes and **float** requires four bytes, the memory layout of a **struct part** might look like this:

Member	Offset
partNo	0
leadTime	14
unitMeas	16
unitCost	22
costQty	26

Storage



Each data type has its own alignment requirement, a requirement (imposed by the central processor unit (CPU) hardware and/or the compiler) that the address of this type of data must be evenly divisible by some number. In the example above, it was assumed that each alignment requirement is no more restrictive than "even address" (divisible by two). Thus, the implementation above occupies 28 bytes. However, such an assumption does not promote portable code because not all hardware and/or compiler alignment requirements (or even storage requirements) are the same.

Therefore, the numeric values of structure offsets may not be hard-coded. The values may be different in each environment. Refer to members by their symbolic member names only.

3.6.2.2.6.2 Structures for Information Hiding. One of the important uses for C structures is to encapsulate interface information. Consider, for example, the FILE structure provided by the standard I/O library. On many systems, the header **stdio.h** defines the name FILE thus:

```
#define FILE struct _file
struct _file
  {
  char *_cur_ptr;  /* where to get/put the next character: !NULL */
  int  _size;      /* how big is the buffer: (0:INT_MAX) */
  char *_buffer;   /* where is the buffer: char[_size] | NULL */
  int  _rcount;    /* how many more gets left: (0:_size) */
  int  _wcount;    /* how many more puts left: (0:_size) */
  char _status;    /* file state: bits */
  char _fd;        /* file descriptor: (0:_NFILES-1) */
  };
```

Thus, the name FILE becomes a synonym for the type **struct _file**. The tag name (_file) and each of the member names (_cur_ptr, _size, etc.) are given names that start with a leading underscore. Names with leading underscore may only appear in code that is privy to the internal details of the associated data structure, not in "user-level" portable code.

The main advantage of this defined-type approach is that the internal details of the representation are somewhat more "hidden" from the functions that use the type. This allows the internal details to be tailored to each particular target system, or to be changed to incorporate more efficient algorithms when needed. Furthermore, those functions that do not examine the internal representation of the object need no source code changes if the object is changed from a structure to a scalar (or to a union).

Use the "leading underscore" name format for tag and member names if the internal details of the structure are not to be inspected by functions outside of the package. Conversely, leading underscore may not be used if the details of the structure are available for inspection by functions that use the structure.

3.6.2.2.6.3 Properties of Structures. In the absence of a specific defining property, a structure is well-defined if all its scalar constituents are well-defined. Referring back to definition of the PART structure (see subparagraph 3.6.2.2.6.1), in order for a PART to be well-defined, all of the following must be true:

partNo	must be a (null-terminated) string
leadTime	must be within the range (0:99)
unitMeas	must contain the string "each", "lb", or "box"
unitCost	must be within the range (0.00:9999.99)
costQty	must be within the range (0:9999)

One of the common abbreviations allowed by C is the initialization of a structure to "all zeros":

```
PART part1 = {0};
```

Thus, if a structure is not well-defined when initialized to zero, document that fact in a comment. However, it must be noted that the program would be much simpler if the members are defined such that the zero-initialized structure is well-defined.

3.6.2.2.6.4 Bit-Fields. C provides a storage-compaction capability for structure members in which each member occupies only a specified number of bits. Such a member is known as a **bit-field**. Bit-fields are useful for reducing the storage needed for a large array of structures and are also quite useful for defining various hardware interfaces which specify the individual bits within a machine word.

The following representation may be used for the time-of-day in hours, minutes, seconds and milliseconds:

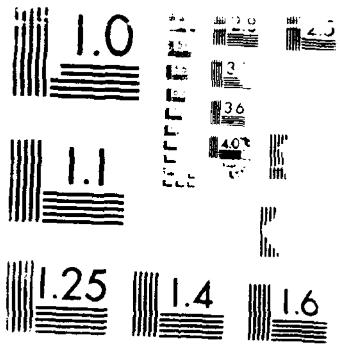
```
TimeOfDay.h(#1):
  /* TimeOfDay.h - bit-field structure for hh:mm:ss.fff */
  #ifndef TimeOfDay_h
  #define TimeOfDay_h
  typedef struct timeOfDay
    {
    unsigned h1 : 2;    /* tens digit of hours       (0:2) */
    unsigned h2 : 4;    /* units digit of hours      (0:9) */
    unsigned m1 : 3;    /* tens digit of minutes     (0:5) */
    unsigned m2 : 4;    /* units digit of minutes    (0:9) */
    unsigned s1 : 3;    /* tens digit of seconds     (0:5) */
    unsigned s2 : 4;    /* units digit of seconds    (0:9) */
    unsigned f1 : 4;    /* first digit of fraction   (0:9) */
    unsigned f2 : 4;    /* second digit of fraction  (0:9) */
    unsigned f3 : 4;    /* third digit of fraction   (0:9) */
    } timeOfDay;
  #endif
```

Each member (bit-field) is declared to be unsigned (int) since this is the only bit-field type that is guaranteed to be portable to all current compilers. Each member is declared to have only as many bits as are necessary to represent the possible digits at its position in the time representation. Representing **h1** takes only two bits to represent the possible values (0, 1, and 2). The largest members need only four bits to represent ten digits (0 through 9). Thus, the total number of bits is 32.

Consecutive bit-field members are allocated by the compiler to the same int-sized word, as long as they fit completely. Thus, on a 32-bit machine, a **timeOfDay** object will occupy exactly one int-sized word. On a 16-bit machine, the first five members (totalling 16 bits) will fit into one int-sized word. and the last four members will fit into an immediately following word. Such an exact fit is rare, however. Add another member such as "day-of-year" to the structure, and the nice size-fitting property disappears. Thus, bit-fields are useful for storage-saving only if they occupy most or all of the

AD-A231 354   SOFTWARE STANDARDS AND PROCEDURES MANUAL FOR THE JNGG   2/2
GRAPHICS PROGRAM(U) JOINT DATA SYSTEMS SUPPORT CENTER
WASHINGTON DC D W HALL 01 DEC 90 DCA/JDSSC-TM-405-90
UNCLASSIFIED   XD-DCA/JDSSC   F/G 12/5   NL

END
FILMED
DTIC


1.0
2.5
3.6
4.0
1.1
1.25
1.4
1.6

space on an **int**, and if the storage-saving property is to be reasonably portable, they must occupy most of the space in a 32-bit integer.

The order of allocation within a word is different in different implementations. Some implementations are "right-to-left" where the first member occupies the low-order position of the word (most PDP-11 and VAX compilers) but most other implementations are "left-to-right."

Thus, to ensure code portability, do not depend upon the allocation order of bit-fields within a word.

3.6.2.2.6.5 Pointers to Structures. Passing and returning structures can cost considerable CPU time since the entire structure is copied each time. Often it is more efficient to pass a pointer to a structure. The declaration

```
struct part *partPtr;
```

declares that **partPtr** points to **struct part**'s. To access the members of the structure that **partPtr** points to, use the "arrow" (->) operator, as shown below:

```
partPtr -> lead_time
```

As discussed in subparagraph 3.6.2.2.5.2, a pointer value can be either undefined (not valid in any pointer contexts), or defined (either NULL or pointing to a valid object of the proper type). If a pointer points to a well-defined structure, the pointer is well-defined.

Since one of the common uses of pointers to structures is as function parameters, their reliable usage is of prime importance to portability. As a simple case, an "out" pointer (used only to modify the pointed-to object) does not care what properties the object has when its address is passed to the function. It is usually an error t pass NULL to an "out" parameter, since it does not point to any object (a NULL parameter could be used to mean "do not store anything this time," but an explicit comment must be given). An "in" pointer (used only to read values from the pointed-to object) should be passed to the address of a well-defined" structure. If, however, the structure is supposed to have some other defined property, the parameter declaration must so indicate in a comment. The same conditions apply to an "in-out" pointer (used both to read values from, and change values in, the pointed-to structure).

Thus, for parameters which are pointers to structures, an "out" pointer parameter is assumed to be non-NULL, pointing to the storage for a structure of the specified type. "In" and "in-out" pointer parameters are assumed to point to a well-defined structure of the specified type. Any exceptions to these assumptions must be noted in a comment on the parameter declaration.

3.6.2.2.7 Dynamic Storage Allocation. Data structures which dynamically grow and shrink as the computation progresses are provided through two functions: **malloc** and **calloc**. If a request for allocated memory asks for more bytes than are available in the heap, either function returns a NULL pointer. Thus, the returned pointer must always be tested to ensure that it is non-NULL.

3.6.2.2.7.1 Freed Storage. When allocated storage has been used and is no longer needed, it can be returned to the heap by the **free** function:

```
free(px);
```

The above example returns to the heap the storage which **px** is pointing to. Note that each time storage is allocated, the allocation functions record internally the size that was given to this allocation. When **free** is called, this internally recorded size is used to determine the amount of storage that is being given back.

One of the important reliability aspects of dynamic allocation is the avoidance of dangling pointers. After calling **free(px)**, the pointer **px** should be considered to have an undefined value. The pointer does still contain a valid machine address but the storage that it points to may subsequently be allocated to some other use. One useful style convention is to immediately assign NULL to a pointer after passing it to the **free** function:

```
free(px);
px = NULL;
```

In some environments, this will ensure that any further access using **px** will generate an execution error. In any environment, it will produce a warning that **px** should not be used to access data.

Since more than one pointer may be pointing to the freed storage, the programmer must determine how many pointers are pointing into the freed storage when a pointer **p** is passed to the **free** function (this number is known as the "reference count" of the storage). Steps must be taken (such as assigning NULL) to ensure that none of these pointers are subsequently used to access the freed storage.

3.6.2.2.7.2 Dead Storage. The continued use of a pointer to freed storage (dangling pointer) has a complementary problem known as **dead storage**--the failure to **free** a chunk of storage when it is no longer needed. A few isolated cases of dead storage are unlikely to be noticed; the storage available is simply decreased. But as the dead storage begins to accumulate, the odds increase that the program will run out of allocatable storage.

Therefore, for every instance in which a programmer allocates storage, there must be a corresponding call to the **free** function to return that storage to the heap.

3.6.2.2.8 Opening Named Files. To perform text-based I/O on files opened by name, the function **fopen** is used:

```
FILE *fp;

fp = fopen("input.dat", "r");
```

The above example opened the file **input.dat** for reading ("r") and the variable **fp** was declared as a pointer to FILE. After the **fopen** call, **fp** will point to a FILE properly initialized for reading. If the open fails, **fp** will contain a NULL pointer.

Thus, always test the returned value from **fopen** to ensure that the open succeeded.

However, should the I/O required of a named file be binary (non textual) in nature, the file should be opened with a call to the **open** function. In this manner, file access is significantly faster because the overhead associated with text-based I/O is bypassed.

3.6.2.2.9 Clean Compilations. C code developed for projects in the JNGG Graphics Program are required to compile and link error-free and warning-free.

3.6.2.3 Format Guidelines. The following subparagraphs specifically delineate the guidelines to be used in C source code generation. These guidelines are grouped into the following topics: lexical elements; declarations and types; names and expressions; statements; functions; and files.

3.6.2.3.1 Lexical Elements. The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate lexical elements. Each lexical element is either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character literal, a string literal, or a comment. Each lexical element must fit on one line, since the end of a line is a separator.

3.6.2.3.1.1 Indentation. The standard indentation is two spaces.

3.6.2.3.1.2 Character Set. Full use should be made of the ISO character set where available. Alternate character replacements should only be used when the corresponding graphical symbols are not available.

3.6.2.3.1.3 Uppercase/Lowercase. Reserved words should appear in lowercase. Type, user-specified variable names, and enumeration value identifiers should adhere to the naming rules specified in subparagraphs 3.6.2.3.1.4 and 3.6.2.3.3.1.

3.6.2.3.1.4 Identifiers. When specifying identifiers in source code, the following rules must be utilized:

a. Identifier names should be meaningful and easily distinguishable from each other, except possibly for loop parameters, array indices, and common mathematical variables, which may be as short as only one character.

b. In **#define**'s, distinct words in identifiers should always be separated by underscores "_".

c. The use of abbreviations in identifiers should be avoided. When used, an abbreviation should be significantly shorter than the word it abbreviates, and its meaning should be clear. The same abbreviations should be used consistently throughout a project and <u>must always be approved by the contractor Project Manager in advance of its usage (final approval by the Government Project Officer)</u>.

3.6.2.3.1.5 <u>Spaces</u>. Single spaces should be used consistently between lexical elements to enhance readability.

3.6.2.3.1.6 <u>Blank Lines</u>. Blank lines should be used to group logically related lines of text and a blank line should always follow a construct whose last line is not at the same indentation level as its first line.

3.6.2.3.1.7 <u>Continuations</u>. Statements extending over multiple lines should always be broken BEFORE reserved words, operator symbols, or one of the following symbols:

: | => .. :=

but they should be broken AFTER a comma (","). Unless otherwise specified in later guidelines, all the continuation lines should be indented at least two levels -- this is, four spaces -- with respect to the original lines they continue.

Long strings extending over more than one line should be broken up at natural boundaries, appropriate to the meaning of the contents of the string, if any.

3.6.2.3.1.8 <u>Comments</u>. Comments should begin with the "/*" aligned with the indentation level of the code that they describe, or to the right of the code, aligned with other such comments.

3.6.2.3.2 <u>Declarations and Types</u>. The following subparagraphs provide format guidelines relevant to commenting, indentation, enumeration types, and object declarations.

3.6.2.3.2.1 <u>Commenting</u>. Type declarations (or groups of declarations) should be commented to indicate what is being defined, if that is not obvious from the type declaration itself.

Object declarations should be commented if the object definition is unclear from the object and type identifiers alone. Note that those properties of an object obtained from its type should not be repeated in comments on the object declaration.

3.6.2.3.2.2 Indentation. All declarations in a single declaration part should begin at the same indentation level.

3.6.2.3.2.3 Enumeration Types. Long enumeration type declarations should be formatted into easily readable columns.

3.6.2.3.2.4 Object Declarations. Object declarations should utilize the conventional format described in subparagraph 3.6.2.2.4.1.1 for declaring variables. Further, they should utilize comments in accordance with the guidelines of subparagraphs 3.6.2.2.4.1.3 and 3.6.2.3.2.1.

All such declarations textually grouped together or appearing as components in a record structure (see subparagraph 3.6.2.2.6.1) should have their <declarators>, "=", and "/*" symbols aligned.

3.6.2.3.3 Names and Expressions. Objects should usually be nouns or noun-phrases that describe the object's qualities or purpose. The following subparagraphs discuss names, parentheses, and continuation lines.

3.6.2.3.3.1 Names. Names which are all uppercase and contain underscores to separate words shall be used to denote **#define** or any **macro**:

```
#define BUTTON_SPACING 10
```

Variable names shall have initial lowercase, capitalize the first letter of the remaining words, and contain no underscores:

```
Widget databaseList;
int    numberButtons;          /* Number of buttons counter */
```

3.6.2.3.3.2 Parentheses. Syntactically redundant parentheses should generally be used to enhance the readability of expressions, especially by indicating the order of evaluation.

3.6.2.3.3.3 Continuation. When a long expression is broken over more than one line, it should be broken near the end of the line before an operator symbol with the lowest reasonable precedence.

3.6.2.3.4 Statement Sequences. Blank lines should be used liberally to break sequences of statements into short, meaningful groups.

3.6.2.3.5 Functions. A function is an independent set of statements for performing some computation.

3.6.2.3.5.1 Function Names. Except as indicated below, a function name should be an imperative verb phrase describing its action. The following guidelines shall be used when naming a function:

a. Names shall have initial uppercase letters, capitalize the first letter of each word, and contain no underscores.

```
ObtainNextToken
IncrementLineCounter
MakeNewGroup
```

b. All functions shall have an explicit type.

c. If the function is a "callback," it shall be a noun phrase and end with the capitalized letters "CB." An exception to this guideline is that "callback" functions may assume the labels of the menu selections needed to access that function and end with the "CB" designator. This exception invariably leads to a verb phrase name.

```
FileDeletionOkCB
OpenQueryCancelCB
MagicExitCB
```

d. Functions that create widgets shall begin with "Create" as the first word; other functions should avoid using "Create" as the first word.

```
Widget    CreateDatabaseList ();
void      NotDoneCB ();
```

e. Non-BOOLEAN valued function names may be noun phrases.

```
TopOfStack
TheComponent
Successor
SensorReading
```

f. BOOLEAN valued functions should have predicate-clause names.

```
StackIsEmpty
LastItem
DeviceNotReady
```

Files names must specify a name in a format acceptable to the host system, and systems differ in their naming rules for files. In general, a name in the form xxxxxx.xxx, where x is a letter or a digit, will be acceptable to most C environments. Within this length and composition constraint, the name should strive to represent the program unit name to the maximum extent allowable.

3.6.2.3.5.2 <u>Function Header</u>. Each function should be preceded by a CSU header comment block containing the documenting information as described and shown in example form in figure 3-4.

3.6.2.3.5.3 <u>Function Definitions</u>. Function definitions should have one of the following formats:

```
<function name:>
  <type mark> <function name> (<param spec>)
--------------------------------------------------------------------
<function name:>
  <type mark> <function name> (<param spec>) <params decl>
--------------------------------------------------------------------
<function name:>
  <type mark> <function name> (<param spec>) <params decl> <block>
```

If the function does not return any value, the <type mark> must be specified as void. If it does return a value, that value must be a scalar. C does not allow a function to return an array, for example. Beyond the mandatory usage of the <type mark>, precision regarding that type is also important, especially in relation to the size of the data being returned.

Guidelines pertaining to the <function name> have been previously discussed in subparagraph 3.6.2.3.5.1.

The <param spec> is the optional part of the syntax where parameters are declared--the local variables of the function which contain the argument values when they are passed. If utilized, the *parameter specifications must* appear in parentheses.

The optional <block> clause completes the function definition. It is comprised of a left-brace, zero or more declarations, zero or more statements, and a right-brace. If this function calls other functions, there should be a declaration for each of these called functions, such as **exp** and **log** shown below. This means that the type of a function is always declared in two separate places--once in the calling function, to say what type it expects from the called function, and once in the definition of the called function, to say what type it will indeed return.

An example of a function definition appears below:

```
pow:
  /* pow - return (positive) x to the power y */
  double pow(x, y)
    double x;  /* base */
    double y;  /* exponent */
    {
    double exp();  /* exponential function */
    double log();  /* natural log function */
```

```
/* **********************************************************************/
/* Unit Name: CreateBgChoiceForm                                        */
/*                                                                      */
/* CSU Id:                                                              */
/*                                                                      */
/* Input:                                                               */
/*   parent - the widget ID of this form's parent.                      */
/*                                                                      */
/* Processing:                                                          */
/*   This CSU creates the form widget for the Bg choice screen.         */
/*                                                                      */
/* Output:                                                              */
/*   bgChoiceForm - global, the widget ID of the insert group form.     */
/*                                                                      */
/* ******************************************************************** */
```

Figure 3-4. C Language Header Comment Block (CSU)

```
return (exp(log(x) * y));
}
```

3.6.2.3.6 Files. Files allow the specification of groups of logically related entities. In their simplest form, files specify pools of common object and type declarations. More generally, files can be used to specify groups of related entities including functions (through prototypes, that can be called from outside the file, while their inner workings remain concealed and protected from outside users.

3.6.2.3.6.1 File Names. A file name should be a noun phrase describing the abstract entity modeled by the file, or simply whatever is being packaged. The file name is also the CSC name for configuration management purposes. The following are some examples:

StackHandler
VehicleController
TerminalOperations
ParserTypes
UtilitiesPackage

3.6.2.3.6.2 File Header. Each file should be preceded by a C language CSC header comment block as shown in figure 3-5.

3.6.3 FORTRAN Language Coding Specifications. New FORTRAN-based source code (either new applications, enhancements, or maintenance-based patch code) shall comply (wherever possible) with the provisions and guidelines of JDSSC's FORTRAN Programming Standards (JDSSC TM 402-90).

3.6.4 General Language Coding Specifications. This subparagraph specifies default design and coding standards to be used when Ada, C, and FORTRAN are not the programming languages being used to develop code.

3.6.4.1 Higher Order Language (HOL). All code shall be written in the HOL specified in the appropriate specification document:

a. Functional Description (FD)

b. System/Segment Specification (SSS)

c. Software Requirements Specification (SRS).

If one or more compilers are specified in the specifications listed above, then all code shall be compiled by the specified compiler(s). Otherwise, all code shall be compiled by the compilers described in the appropriate SDP.

If the HOL does not contain the control constructs of subparagraph 3.6.4.2, the pre-compiler (if any) specified in the appropriate SDP shall be used. If a pre-compiler which is acceptable to JNGG does not exist, then these control constructs shall be simulated (i.e., code in the language used shall follow

```
/* ******************************************************************* */
/* File Name (CSC) : BgChoiceOperations                                 */
/*                                                                      */
/* CSC Id          :                                                    */
/*                                                                      */
/* Author          : Geoff Raines                                       */
/*                                                                      */
/* Create Date     : 05/11/90                                           */
/*                                                                      */
/* Revision History:                                                    */
/*   06/30/90 GR <Summarize exactly what changes were done and why>.    */
/*                                                                      */
/* Purpose:                                                             */
/*   Explain what this unit does to support the CSCI.                   */
/*                                                                      */
/* Headers Accessed:                                                    */
/*   <stdio.h> - needed for the definition of FILE                      */
/*   <ctype.h> - needed for the routine isspace()                       */
/*   "UnixFileSystemTools.h" - needed for the routine Determine Inode   */
/*      Type and its definitions.                                       */
/*                                                                      */
/* Externed Objects Used:                                               */
/*   magicShell - this Widget is used to attach the Widgets created in */
/*                Bg.                                                   */
/*                                                                      */
/* Hardware Dependencies:                                               */
/*   List the access type, purpose, and justification for any,          */
/*   such as: registers, memory, bulk storage, ports, and machine       */
/*   code.                                                              */
/*                                                                      */
/* Compiler Dependencies:                                               */
/*   List any special requirements and their justification, such        */
/*   as: special pragmas, optional implementation, and specific         */
/*   compilers.                                                         */
/*                                                                      */
/* ****************************************************************** */
```

Figure 3-5. C Language Header Comment Block (CSC) the logic of the constructs without explicitly using their names in the code).

If language simulation is used, the same form of the simulated constructs shall be uniformly applied throughout the code.

A waiver from JNGG Branch Management shall be required in order to write code in assembly language or in some HOL other than the HOL specified in the documents listed above.

3.6.4.2 Control Constructs. Code shall be written using only the following control constructs:

a. SEQUENCE

b. IF-THEN-ELSE

c. DO-WHILE

d. DO-UNTIL

e. CASE.

These control constructs refer to the control logic within a CSU/program while it is executing and do not preclude the calling or passing of processor control to other CSUs/programs (e.g., exception handlers, interrupt service routines).

3.6.4.3 Modularity. The source code for each CSU/program shall not exceed 200 executable, non-expandable statements. Additionally, CSUs/programs shall exhibit the following characteristics:

a. Local variables within different CSUs/programs shall not share the same storage locations.

b. Each CSU/program shall perform a single function.

c. Modification of a CSU's/program's code during execution shall be prohibited.

d. Each CSU/program shall be uniquely named.

e. All CSUs/programs shall follow a standard format consisting of prologue, declarative statements, and executable statements or comments, in that order.

f. Except for error exits, each CSU/program shall have a single entry point and a single exit point.

g. Coding style conventions shall be consistent among all CSUs/programs.

3.6.4.4 Symbolic Parameters. To the maximum extent practical, symbolic parameters shall be used, in lieu of specific numeric values, to represent constants, relative location within a table, and size of data structure.

3.6.4.5 Naming. Naming conventions shall be uniform throughout the CSCI/module and shall employ meaningful names which clearly identify the constant, variable, function performed, and any other objects used in the CSCI/module, to a reader of the source code. Language keywords shall not be used as identifiers.

3.6.4.6 Mixed-Mode Operations. Mixed-mode operations shall be avoided (e.g., arithmetic between real numbers and integer numbers). However, if it is necessary to use them, they shall be clearly identified and described using prominent comments within the source code.

3.6.4.7 Paragraphing, Blocking, and Indenting. Paragraphing, blocking by blank lines, and indenting shall be used to enhance the readability of the code. The indenting factor used shall be two spaces.

3.6.4.8 Complicated Expressions. Compound negative Boolean expressions shall be prohibited. Nesting beyond five levels should be avoided.

3.6.4.9 Compound Expressions. The order of evaluation for compound expressions shall be clarified through the use of parentheses and spacing.

3.6.4.10 Single Statement. Each line of source code shall contain, at most, one executable statement.

3.6.4.11 Comments. Comments shall be set off from the executable source code in a uniform manner. Before each CSU's/program's executable section, a prologue section shall describe the following details:

a. The CSU's/program's purpose and how it works

b. Functions, performance requirements, and external interfaces of the CSCI/module that the CSU/program helps implement

c. Other CSUs/programs (subroutines, procedures, functions) called and the calling sequence

d. Inputs and outputs, including data files referenced during CSU/program entry or execution (for each referenced file, the name of the file, usage (input, output, or both), and a brief summary of the purpose for referencing the file)

e. Use of global and local variables and, if applicable, registers and memory locations

f. The identification of special tasks that are internally defined, and the size/structure of which are based on external requirements

g. The programming department or section responsible for the CSU/program

h. Date of creation of the CSU/program

i. Date of latest revision, revision number, problem report number, and title associated with the revision.

3.6.4.12 Error and Diagnostic Messages. To the maximum extent practical, all error and diagnostic messages shall be presented in a uniform manner and shall be self-explanatory. They shall not require the operator to perform table look-ups or further processing of any kind to interpret the message.

3.6.5 Programming Languages and Graphics Standards. All software development is to be accomplished using the Ada, C, or FORTRAN programming languages. All graphics processing will utilize the Graphical Kernel System (GKS), X Windows/Motif, Microsoft Windows, or the Programmer's Hierarchical Interface Graphics System (PHIGS). Support for metafile conversions to Data Interchange Format (DIF) format will also be supported. Specific waivers to these programming language and graphics standards must be approved by the Configuration Control Board (CCB).

3.6.6 Diagramming Symbology and Standards. The use of symbology to graphically depict the control flow, data flow, and hierarchical structure of software development is critical to ease understanding of the process both from a developer's standpoint as well as the Government. To aid and support this goal, a standardized set of approved symbols and the methodology behind their use is paramount. This section will accomplish that purpose.

Software development for projects comprising the JNGC Graphics Program will make use of two separate classes of symbology:

a. Gane and Sarson approach (tailored)

b. Booch diagrams.

The tailored Gane and Sarson style is well-suited to depict control and data flow relationships between CSCIs, CSCs, or even CSUs. Based upon a limited set of simple, easily-remembered symbols, this approach offers a clear and concise, yet elegant, method to graphically show such information. The standard set of metasymbols to be used for this approach is illustrated in figure 3-6.

Alternately, Booch diagrams are an excellent tool for graphically depicting programmatic design structures in the context of an Ada-based topology. The standard set of metasymbols to be used for Booch diagrams is a far richer set and illustrates both the concept of a specific program unit as well as something about its design. For instance, a CSCI in the design may be conceived as an Ada package structure which, in Booch diagrams, can be depicted as two symbols to illustrate both the specification and body portions

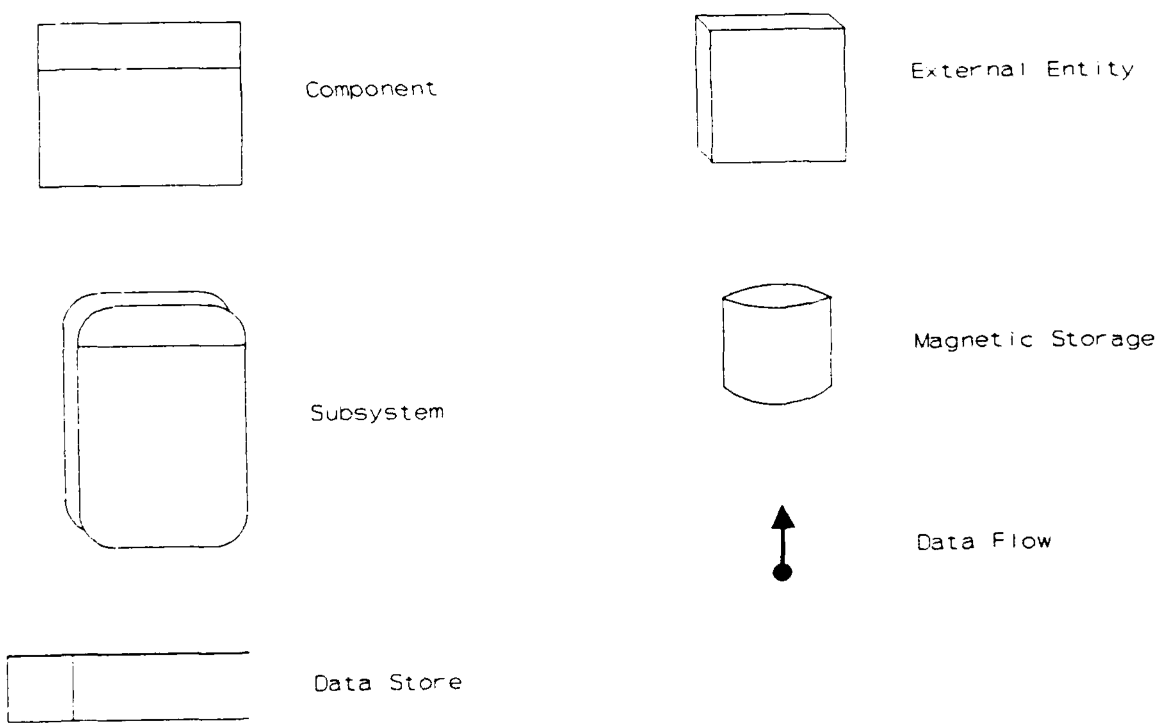


Figure 3-6. Gane and Sarson Metasymbols

of that Ada construct. Figure 3-7 shows the standard set of Booch symbols to be used.

3.6.7 Documentation Standards. All documentation developed and delivered to the Government for the JNGG Graphics Program shall conform to the guidelines of the appropriate Data Item Description (DID), the appropriate development standard (e.g., DOD-STD-7935A), and JDSSC PM 1-90 specifications. Printed documentation shall be delivered in letter quality (laser printed is preferable) using the Prestige Elite 12 pitch font and will be left-justified.

Final versions of documentation shall be delivered in both hardcopy version as noted above and in WordPerfect 5.1 format on a 5.25 inch floppy diskette (DOS-formatted) unless specifically tasked otherwise by a Statement of Work (SOW), a Task Order (TO), or by the Contracting Officer's Representative (COR).

3.7 Formal Reviews

The JNGG Graphics Program will utilize a number of formal reviews and audits the choice of which is relative to the type of software development being conducted: full life cycle or rapid application development (RAD). Full life cycle development is defined as being in accordance with the life cycles described in DOD-STD-7935A or DOD-STD-2167A. This type of software development will use the formal reviews and audits of MIL-STD-1521B:

a. System Design Review (SDR)

b. Software Specification Review (SSR)

c. Preliminary Design Review (PDR)

d. Critical Design Review (CDR)

e. Test Readiness Review (TRR)

f. Functional Configuration Audit (FCA)

g. Formal Qualification Review (FQR)

h. Physical Configuration Audit (PCA).

Software development that is RAD-based is designed to be more flexible and may utilize any and all of the formal reviews/audits noted above. In many cases, however, it may be more advantageous to capture the flexibility of the process through a series of builds all being integrated into the predecessor and using the following formal reviews:

a. In-Process Review (IPR)

b. Initial Operational Capability (IOC)

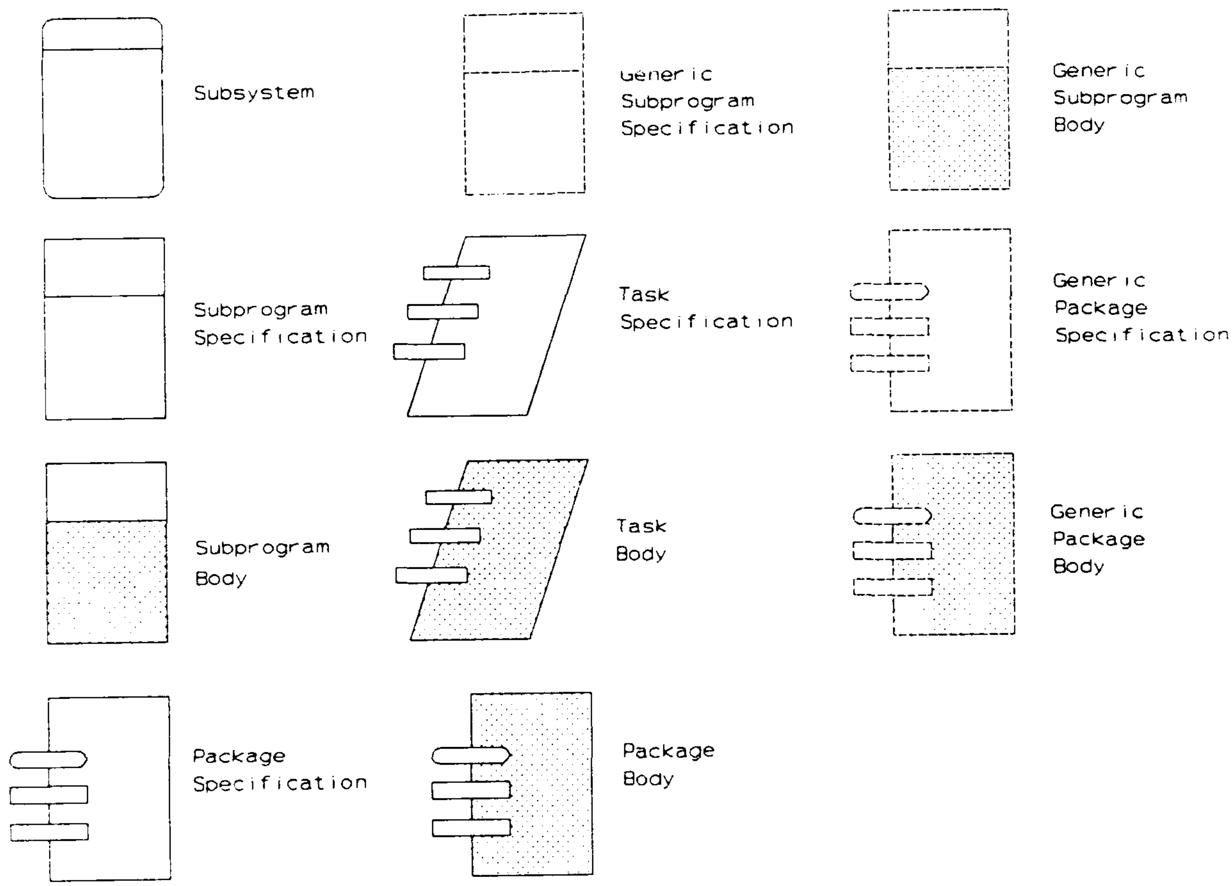


Figure 3-7. Booch Diagram Metasymbols

c. Final Operational Capability (FOC).

Agendas and minutes for all reviews and audits will be provided by using the following DIDs for format and content guidelines:

a. DI-A-3029, Agenda - Design Reviews, Configuration Audits and Demonstrations

b. DI-E-3118, Minutes of Formal Reviews, Inspections and Audits.

SECTION 4. NOTES

This section contains information of general interest which aids in understanding this specification. Specifically, bibliography references to include both source and issue date are provided as well as a terms and abbreviations paragraph.

4.1 Bibliography

The following references were used in the preparation of this document:

a. American National Standards Institute (ANSI), Graphical Kernel System (GKS) Functional Description, ANSI Standard X3.124-1985, New York, NY, 24 June 1985

b. ANSI, IEEE Recommended Practice for Ada As a Program Design Language, ANSI/IEEE Std 990-1987, New York, NY, 1 October 1987

c. ANSI, Programming Language C, ANSI Standard X3.159-1989, New York, NY, 16 December 1989

d. ANSI, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, 17 February 1983

e. Booch, Grady, Software Engineering with Ada, Second Edition, ISBN 0-8053-0604-8, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987

f. Bruce, Phillip and Pederson, Samuel M., The Software Development Project: Planning and Management, New York, NY, 1982

g. Charette, Robert N., Software Engineering Environments: Concepts and Technology, ISBN 0-07-010645-2, Intertext Publications, Inc., New York, NY, 1986

h. Data Interchange Format (DIF) Clearinghouse, DIF Technical Specification, DIF-0283, Newton Lower Falls, MA, 1983

i. Department of Defense (DOD), Defense System Software Development, Department of Defense Standard DOD-STD-2167A, Washington, D.C., 29 February 1988

j. DOD, Defense System Software Quality Program, Department of Defense Standard DOD-STD-2168, Washington, D.C., 29 April 1988

k. DOD, DOD Automated Information Systems (AIS) Documentation Standards, Department of Defense Standard DOD-STD-7935A, Washington, D.C., 31 October 1988

l. DOD, Software Standards and Procedures Manual, Data Item Description (DID) DI-MCCR-80011, Washington, D.C., 4 June 1985

m. Joint Data Systems Support Center (JDSSC), Documentation Standards and Publications Style Manual, Procedures Manual (PM) 1-90, Washington, D.C., 1 August 1990

n. JDSSC. FORTRAN Programming Standards, Technical Memorandum (TM) 402-90, Washington, D.C., 1 November 1990

o. JDSSC, Software Metrics Program, Procedures Manual (PM) 4-90, Washington, D.C., 1 November 1990

p. JDSSC, Standards and Procedures for Software Projects, Procedures Manual (PM) 2-90, Washington, D.C., 1 September 1990

q. Martin, James, Recommended Diagramming Standards for Analysts & Programmers: A Basis for Automation, ISBN 0-13-767377-9, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987

r. National Aeronautics and Space Administration (NASA), Ada Style Guide (Version 1.1), SEL-87-002, Greenbelt, MD, May 1987

s. NASA, Assessing the Ada Design Process and its Implications: A Case Study, SEL-87-004, July 1987

t. National Institute of Standards and Technology (NIST), POSIX: Portable Operating System Interface for Computer Environments, Federal Information Processing Standards Publication (FIPS PUB) Number 151, 12 September 1988

u. Plauger, P.J. and Brodie, Jim, Standard C, ISBN 1-55615-158-6, Microsoft Press, Redmond, WA, 1989

v. Plum, Thomas, C Programming Guidelines, ISBN 0-911537-03-1, Plum Hall, Inc., New York, NY, 1984

w. Plum, Thomas, Reliable Data Structures in C, ISBN 0-911537-04-X, Plum Hall, Inc., New York, NY, 1985

x. Pressman, Roger S., Software Engineering: A Practitioner's Approach, Second Edition, ISBN 0-07-050783-X, McGraw-Hill Book Company, New York, NY, 1987

y. United States Air Force (USAF), Agenda - Design Reviews, Configuration Audits and Demonstrations, Data Item Description (DID) DI-A-3029/S-105-1, Washington, D.C., 21 May 1971

z. USAF, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, Military Standard MIL-STD-483A, Washington, D.C., 4 June 1985

aa. USAF, Minutes of Formal Reviews, Inspections and Audits, Data Item Description (DID) DI-E-3118/C-131-1, Washington, D.C., 26 February 1971

ab. USAF, Technical Reviews and Audits for Systems, Equipments, and Computer Software, Military Standard MIL-STD-1521B, Washington, D.C., 4 June 1985

ac. United States Department of Commerce, National Technical Information Service (NTIS), The X Window System, MIT Laboratory for Computer Science Technical Report Number MIT/LCS/TR-368, Cambridge, MA, November 1986

ad. United States Navy (USN), A Tailoring Guide for DOD-STD-2167A, Defense System Software Development, Military Handbook MIL-HDBK-287, Washington, D.C., 11 August 1989

ae. USN, Configuration Control - Engineering Changes, Deviations and Waivers, Military Standard Number MIL-STD-480B, Washington, D.C., 15 July 1988

af. USN, Configuration Control - Engineering Changes (Short Form), Deviations and Waivers, Military Standard MIL-STD-481B, Washington, D.C., 15 July 1988

ag. USN, Configuration Status Accounting Data Elements and Related Features, Military Standard MIL-STD-482A, Washington, D.C., 1 April 1974.

4.2 Terms and Abbreviations

The following terms, abbreviations, and acronyms specific to this document are listed below.

Ada ----------- The Ada computer language defined by ANSI/MIL-STD-1815A and directed to be the DOD programming language
ADP ----------- Automated Data Processing
ANSI ---------- American National Standards Institute
CBSI ---------- Computer Based Systems, Inc.
CCB ----------- Configuration Control Board; the focal point for coordination of all JNGG Graphics Program enhancement and change activity
CCI ----------- Configuration Control Item
CDR ----------- Critical Design Review as specified by MIL-STD-1521B
CDRL ---------- Contract Data Requirements List
CIA ----------- Central Intelligence Agency
CINCPAC -------- Commander-in-Chief, Pacific

Class I Change - An engineering change which affects the contractually specified form, fit, or function of a configuration item as defined by MIL-STD-480B.

Class II Change --------- An engineering change which does not fall within the definition of a Class I change such as: a change to documentation only or a change in hardware which does not affect any Class I determination factor as specified in MIL-STD-480B.

CM ------------- Configuration Management

Configuration Identification - The current approved or conditionally approved technical documentation for a configuration item as set forth in specifications, drawings, and associated lists, and documents referenced therein

Configuration Item ----------- Hardware or software, or an aggregation of both, which is designated by the contracting agency for configuration management

COR ------------ Contracting Officer's Representative

COTS ----------- Commercial Off-The-Shelf

CSC ------------ Computer Software Component; a functional or logically distinct part of a CSCI which can be further subdivided into additional CSCs and/or CSUs

CSCI ----------- Computer Software Configuration Item

CSU ---- ------- Computer Software Unit

CUC ------------ Common User Contract

DATSEL --------- Data Selection Module of GIPSY

DBMS ----------- Database Management System

DCA ------------ Defense Communications Agency

DCEC ----------- Defense Communications Egineering Center

Development Configuration -- The contractor's software and associated technical documentation that defines the evolving configuration of a CSCI during development; it is under the development contractor's configuration control and describes the software configuration of the design, coding, and testing effort

DI ------------- Data Item

DIA ------------ Defense Intelligence Agency

DID ------------ Data Item Description

DISPLA --------- Business Graphics Module of GIPSY

DOD ------------ Department of Defense

DOD-STD -------- Department of Defense Standard

DOS ------------ Disk Operating System

DMS ------------ DeLorme Mapping System

ETC ------------ Enhanced Terminal Capability

FCA ------------ Functional Configuration Audit as specified in MIL-STD-1521B

FD ------------- Functional Description as defined by DOD-STD-7935A and its associated DID # DI-IPSC-80689

FIPS PUB ------- Federal Information Processing Standards Publication

Formal Test ---- A test conducted in accordance with test plans and procedures approved by the contracting agency and witnessed by an authorized contracting agency representative, to show that the software satisfies a specified requirement

FORTRAN -------- The FORTRAN programming language as defined by ANSI X3.9-1978

GDRMOD --------- Generalized Data Reports Module of GIPSY

GEOMOD --------- Geographic Mapping Module of GIPSY

GFSC ----------- Goddard Space Flight Center in Greenbelt, Maryland

GIPSY ---------- Graphic Information Presentation System; also the Executive Module of the GIPSY System

GKS ------------ Graphical Kernel System as defined by ANSI X3.124-1985

GMPS ----------- GIPSY Metafile Processing Subsystem; a now defunct GIPSY subsystem

GUI ------------ Graphical User Interface

HIS ------------ Honeywell Information Systems

HOL ------------ Higher Order Language (e.g., Ada, FORTRAN, C)

HWCI ----------- Hardware Configuration Item

H6000 ---------- Generic term for the family of Honeywell mainframe computers that include the HIS 6080--the WWMCCS host platform

IBM ------------ International Business Machines, Inc.

IEEE ----------- Institute of Electrical and Electronics Engineers

Informal Test -- Any test which does not meet all requirements of a formal test

IOC ------------ Initial Operational Capability

IPR ------------ In-Process Review

IR ------------- Software Release Incident Report

ISBN ----------- International Standard Book Number

IV&V ----------- Independent Verification and Validation

I/O ------------ Input/Output

JDSSC ---------- Joint Data Systems Support Center

JN ------------- NMCS ADP Directorate

JNG ------------ General Applications Division

JNGG ----------- Information Systems Branch

JOPES ---------- Joint Operation Planning and Execution System

JPM ------------ Joint Program Manager

JPMO ----------- Joint Program Management Office

JSMS ----------- Joint Staff Mapping System

Kbyte ---------- 1,024 bytes of data

LCS ------------ MIT Laboratory for Computer Science

MAGIC ---------- Mapping and Graphic Information Capability

Mbyte ---------- 1,024,000 bytes of data

MCCR ----------- Mission-Critical Computer Resources

MIL-HDBK ------- Military Handbook

MIL-STD -------- Military Standard

MIT ------------ Massachusetts Institute of Technology

MTXGEN --------- Matrix Generation Module of GIPSY

NASA ----------- National Aeronautics and Space Administration

NIST ----------- National Institute of Standards and Technology; formerly the National Bureau of Standards

NMCS ----------- National Military Command System
NORAD ---------- North America Air Defense Command
OSD ------------ Office of the Secretary of Defense
PCA ------------ Physical Configuration Audit as specified in MIL-STD-1521B
PDL ------------ Program Design Language
PDR ------------ Preliminary Design Review as specified by MIL-STD-1521B
PM ------------- Procedures Manual
RAD ------------ Rapid Application Development
RCL ------------ Release Capability List
RP ------------- Release Plan as specified in JDSSC PM 1-90
SCMP ----------- Software Configuration Management Plan as specified in DOD-STD-2167 and its associated DID # DI-MCCR-80009
SDD ------------ Software Design Document as specified in DOD-STD-2167A and its associated DID # DI-MCCR-80012A
SDF ------------ Software Development File; a repository for a collection of material pertinent to the development or support of software
SDL ------------ Software Development Library; a controlled collection of software documentation, and associated tools and procedures used to facilitate the orderly development and subsequent support of software
SDP ------------ Software Development Plan as defined in DOD-STD-2167A and its associated DID # DI-MCCR-80030A
SDR ------------ System Design Review as specified in MIL-STD-1521B
SEL ------------ Software Engineering Laboratory; an organization sponsored by NASA/GSFC
SLOC ----------- Source line of code; an 80-byte record processible by the computer that includes comments, executable code, and non-executable code (i.e., declarations and blank lines)
SOW ------------ Statement of Work
SPCR ----------- Software Problem/Change Report as specified in MIL-STD-483A
SQA ------------ Software Quality Assurance
SQL ------------ Structured Query Language as specified by ANSI X3.135-1986
SQPP ----------- Software Quality Program Plan as specified by DOD-STD-2168 and its associated DID # DI-QCIC-80572
SRS ------------ Software Requirements Specification as defined by DOD-STD-2167A and its associated DID # DI-MCCR-80025A
SSPM ----------- Software Standards and Procedures Manual as defined by DOD-STD-2167 and its associated DID (DI-MCCR-80011)
SSR ------------ Software Specification Review as specified in MIL-STD-1521B
SSS ------------ System/Segment Specification as defined by DOD-STD-2167A and its associated DID # DI-CMAN-80008A
STP ------------ Software Test Plan as specified in DOD-STD-2167A and its associated DID # DI-MCCR-80014A
SYNTAX --------- Syntax Module of GIPSY
TM ------------- Technical Memorandum as specified in JDSSC PM 1-90
TO ------------- Task Order
TPLOT ---------- Terra Plot System
TR ------------- Technical Report
TRR ------------ Test Readiness Review as specified in MIL-STD-1521B
TSR ------------ Technical Support Requirement

Union ---------- A C language structure (struct) in which all the offsets are zero; all members of a union start at byte zero of the object's storage and occupy the same space
USAF ----------- United States Air Force
VIP ------------ Visual Information Projector
WBS ------------ Work Breakdown Structure
WIS ------------ WWMCCS Information Systems
WITS ----------- WWMCCS Intelligent Terminal System; a now defunct system that executed on the WSGT
WSGT ----------- The now defunct WWMCCS Standard Graphic Terminal; the Aydin 5807 processor
WWS ------------ WAM Workstation; previously known as the WIS Workstation

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTION

Addressees	Copies
JDSSC Codes	
JTSA-P (Record and Reference Set)	3
JNGG	45
Defense Technical Information Center (DTIC)	
Cameron Station, Alexandria, VA 22304-6145	2

	50

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
1 December 1990

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
Software Standards and Procedures Manual for the JNGG Graphics Program

5. FUNDING NUMBERS

6. AUTHOR(S)
David W. Hall

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Joint Data Systems Support Center (JDSSC)
Room BF670C, The Pentagon
Washington, D.C., 20301-7010

8. PERFORMING ORGANIZATION REPORT NUMBER
TM 405-90

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This Software Standards and Procedures Manual (SSPM) contains the standards, procedures, guidelines, and restrictions to be used in the development of software for the JNGG Graphics Program.

The major section of the SSPM is section 3 (Software Standards and Procedures). That section is divided into seven major subsections. These subsections include software development tools, techniques, and methodologies (paragraph 3.1); critical lower-level Computer Software Component (CSC) and Computer Software Unit (CSU) selection criteria (paragraph 3.2); software development library (paragraph 3.3); software development files (paragraph 3.4); documentation formats for informal tests (paragraph 3.5); design and coding standards (paragraph 3.6); and formal reviews (paragraph 3.7).

14. SUBJECT TERMS
N/A

15. NUMBER OF PAGES
121

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	None